IMP2 — Simple Program Verification in Isabelle/HOL

Peter Lammich       Simon Wimmer

June 11, 2019

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 4
   1.1 Primitives .......................... 4
   1.2 Arithmetic Expressions .............. 4
   1.3 Boolean Expressions .................. 5
   1.4 Commands ........................... 5
      1.4.1 Minimal Concrete Syntax ........ 6
   1.5 Program ............................ 6
   1.6 Default Array Index ................ 6

2 Semantics of IMP 7
   2.1 State ............................ 7
      2.1.1 State Combination .............. 7
   2.2 Arithmetic Expressions .............. 8
   2.3 Boolean Expressions ................. 8
2.4 Big-Step Semantics .......................... 8
  2.4.1 Proof Automation ......................... 9
  2.4.2 Automatic Derivation ..................... 10
2.5 Command Equivalence ....................... 11
  2.5.1 Basic Equivalences ....................... 11
2.6 Execution is Deterministic ................. 12
2.7 Small-Step Semantics ....................... 12
  2.7.1 Equivalence to Big-Step Semantics ..... 13
2.8 Weakest Precondition ....................... 16
  2.8.1 Basic Properties ......................... 17
  2.8.2 Unfold Rules ............................ 17
  2.8.3 Weakest precondition and Program Equivalence 18
  2.8.4 While Loops and Weakest Precondition 19
2.9 Invariants for While-Loops ................. 20
  2.9.1 Partial Correctness ..................... 20
  2.9.2 Total Correctness ....................... 21
  2.9.3 Standard Forms of While Rules .......... 21
2.10 Modularity of Programs ................... 22
2.11 Strongest Postcondition ................. 22
2.12 Hoare-Triples ............................. 23
  2.12.1 Sets of Hoare-Triples .................. 24
  2.12.2 Deriving Parameter Frame Adjustment Rules 24
  2.12.3 Proof for Recursive Specifications .... 25
2.13 Completeness of While-Rule .............. 26

3 Annotated Syntax .......................... 28
  3.1 Annotations ............................... 28
  3.2 Hoare-Triples for Annotated Commands .... 29

4 Quickstart Guide ......................... 30
  4.1 Introductory Examples ..................... 30
    4.1.1 Variant and Invariant Annotations ... 32
    4.1.2 Recursive Procedures .................. 32
  4.2 The VCG ................................. 33
  4.3 Advanced Features ....................... 33
    4.3.1 Custom Termination Relations ......... 33
    4.3.2 Partial Correctness ................... 34
    4.3.3 Arrays ............................... 34
  4.4 Proving Techniques ...................... 35
    4.4.1 Auxiliary Lemmas ...................... 35
    4.4.2 Inlining ............................. 35
    4.4.3 Functional Refinement ............... 36
    4.4.4 Data Refinement ...................... 36
  4.5 Troubleshooting ......................... 36
    4.5.1 Invalid Variables in Annotations ... 36
    4.5.2 Wrong Annotations .................... 37
    4.5.3 Calls to Undefined Procedures ....... 37
  4.6 Missing Features ....................... 37
    4.6.1 Elaborate Warnings and Errors ...... 37
    4.6.2 Static Type Checking .................. 37
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.

\[ \text{type-synonym } vname = \text{string} \]
\[ \text{type-synonym } pname = \text{string} \]

The variable names are partitioned into local and global variables.

\[ \text{fun is-global :: vname } \Rightarrow \text{bool} \text{ where} \]
\[ \text{is-global } [] \longleftrightarrow \text{True} \]
\[ \text{is-global } (\text{CHR} \quad "G"\#-) \longleftrightarrow \text{True} \]
\[ \text{is-global } - \longleftrightarrow \text{False} \]

\[ \text{abbreviation is-local a } \equiv \neg \text{is-global a} \]

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.

\[ \text{type-synonym } pval = \text{int} \]
\[ \text{type-synonym } val = \text{int } \Rightarrow pval \]

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.

\[ \text{datatype aexp = } \]
\[ N \text{ int} \]


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
| \textit{AssignIdx} \textit{vname} \textit{aexp} \textit{aexp} — Assign to index in array
| \textit{ArrayCpy} \textit{vname} \textit{vname} — Copy whole array
| \textit{ArrayClear} \textit{vname} — Clear array
| \textit{Assign-Locals} \textit{vname} \Rightarrow \textit{val} — Internal: Assign all local variables simultaneously

— Block
| \textit{Seq} \textit{com} \textit{com} — Sequential composition
| \textit{If} \textit{bexp} \textit{com} \textit{com} — Conditional
| \textit{While} \textit{bexp} \textit{com} — While-loop
| \textit{Scope} \textit{com} — Local variable scope

— Procedure
| \textit{PCall} \textit{pname} — Procedure call
| \textit{PScope} \textit{pname} \rightarrow \textit{com} \textit{com} — Procedure scope

1.4.1 Minimal Concrete Syntax

The commands come with a minimal concrete syntax, which is compatible to the syntax of \textit{IMP}.

\textbf{notation} \textit{AssignIdx} (\texttt{[-]} ::= [-1000, 0, 61] 61)
\textbf{notation} \textit{ArrayCpy} (\texttt{[-]} ::= [-1000, 1000] 61)
\textbf{notation} \textit{ArrayClear} (\texttt{CLEAR} \texttt{-[]} [1000] 61)
\textbf{notation} \textit{Seq} (\texttt{;;} / - [61, 60] 60)
\textbf{notation} \textit{If} ((\texttt{IF} / \texttt{THEN} / \texttt{ELSE} -) [0, 0, 61] 61)
\textbf{notation} \textit{While} ((\texttt{WHILE} / \texttt{DO} -) [0, 61] 61)
\textbf{notation} \textit{Scope} (\texttt{SCOPE} - [61] 61)

1.5 Program

\textbf{type-synonym} \textit{program} = \textit{pname} \rightarrow \textit{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.

\textbf{abbreviation} \textit{V} \textit{x} \equiv \textit{Vidx} \textit{x} (\texttt{N} 0)
\textbf{abbreviation} \textit{Assign} \ (- ::= -[1000, 61] 61)
\textbf{where} \textit{x} ::= \textit{a} \equiv (\textit{x}[\texttt{N} 0] ::= \textit{a})
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 ≡ λx. λi. 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 (<> | -| -)
  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 by auto

lemma combine-nest:
  <> s|(t’)|t> = <> s|t>
  <> s|t’|t> = <> s|t>
  by auto

lemma combine-query:
  is-local x ⇒ <> s|t> x = s x
  is-global x ⇒ <> s|t> x = t x
  by auto

lemma combine-upd:

is-local $x \mapsto <s|t>(x:=v) = <s(x:=v)|t>$
is-global $x \mapsto <s|t>(x:=v) = <s|t(x:=v)>
by `auto`

lemma `combine-cases[cases type]`:
`obtains l g where s = <l|g>`
by `(fastforce)`

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) s = f (aval a s)`
| `aval (Binop f a1 a2) s = f (aval a1 s) (aval a2 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 b1 b2) s = f (bval b1 s) (bval b2 s)`
| `bval (Cmpop f a1 a2) s = f (aval a1 s) (aval a2 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 x state => state => bool`
(\_: - \Rightarrow - [1000,55,55] 55)
where
— No-Op
`Skip: \pi: (SKIP, s) \Rightarrow s`

— Assignments
| `AssignIdx: \pi: (x[i] ::= a, s) \Rightarrow s(x := (s x)(aval i s := aval a s))`
<table>
<thead>
<tr>
<th>ArrayCopy: $\pi:(x[] := y,s) \Rightarrow s(x := s y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayClear: $\pi:($CLEAR $x[]),s) \Rightarrow s(x := (\lambda x. 0))$</td>
</tr>
<tr>
<td>Assign- Locals: $\pi:($Assign-Locals $l,s) \Rightarrow \langle l\rangle s$</td>
</tr>
</tbody>
</table>

— Block commands |
| Seq: $\pi:$(c_1,s_1) \Rightarrow s_2; $\pi:$(c_2,s_2) \Rightarrow s_3 $\Rightarrow$ $\pi:$(c_1;c_2, s_1) \Rightarrow s_3$ |
| IfTrue: $\pi:$(bval b s; $\pi:$(c_1,s) \Rightarrow t $\Rightarrow$ $\pi:$(IF b THEN c_1 ELSE c_2, s) $\Rightarrow$ t$ |
| IfFalse: $\pi:$(¬bval b s; $\pi:$(c_2,s) \Rightarrow t $\Rightarrow$ $\pi:$(IF b THEN c_1 ELSE c_2, s) $\Rightarrow$ t$ |
| Scope: $\pi:$(c, <s|>s>) $\Rightarrow$ s’ $\Rightarrow$ $\pi:$(SCOPE c, s) $\Rightarrow$ <s|s’>$ |
| WhileFalse: $\pi:$(¬bval b s $\Rightarrow$ $\pi:$(WHILE b DO c,s) $\Rightarrow$ s$ |
| WhileTrue: $\pi:$(bval b s; $\pi:$(c,s_1) $\Rightarrow$ s_2; $\pi:$(WHILE b DO c, s_2) $\Rightarrow$ s_3 $\Rightarrow$ $\pi:$(WHILE b DO c, s_1) $\Rightarrow$ s_3$ |

— Procedure commands |
| PCall: $\pi:$(p = Some c; $\pi:$(c,s) $\Rightarrow$ t $\Rightarrow$ $\pi:$(PCall p,s) $\Rightarrow$ t$ |
| PScope: $\pi:$(c,s) $\Rightarrow$ t $\Rightarrow$ $\pi:$(PScope p, c) $\Rightarrow$ 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: $\pi:$(SKIP,s) $\Rightarrow$ t  
**inductive-simps** AssignIdx-simp: $\pi:$(x[i] := a,s) $\Rightarrow$ t  
**inductive-simps** ArrayCopy-simp: $\pi:$(x[] := y,s) $\Rightarrow$ t  
**inductive-simps** ArrayInit-simp: $\pi:($CLEAR x[],s) $\Rightarrow$ t  
**inductive-simps** AssignLocals-simp: $\pi:($Assign- Locals l,s) $\Rightarrow$ t  

**inductive-simps** Seq-simp: $\pi:$(c_1;c_2,s_1) $\Rightarrow$ s_2  
**inductive-simps** If-simp: $\pi:$(IF b THEN c_1 ELSE c_2,s) $\Rightarrow$ t  
**inductive-simps** Scope-simp: $\pi:$(SCOPE c,s) $\Rightarrow$ t  
**inductive-simps** PCall-simp: $\pi:$(PCall p,s) $\Rightarrow$ t  
**inductive-simps** PScope-simp: $\pi:$(PScope p, c) $\Rightarrow$ t  

**lemmas** big-step-simps =  
  Skip-simp AssignIdx-simp ArrayCopy-simp ArrayInit-simp 
  Seq-simp If-simp Scope-simp PCall-simp PScope-simp

**inductive-cases** SkipE[elim!]: $\pi:$(SKIP,s) $\Rightarrow$ t  
**inductive-cases** AssignIdxE[elim!]: $\pi:$(x[i] := a,s) $\Rightarrow$ t  
**inductive-cases** ArrayCopyE[elim!]: $\pi:$(x[] := y,s) $\Rightarrow$ t  
**inductive-cases** ArrayInitE[elim!]: $\pi:$(CLEAR x[],s) $\Rightarrow$ t  
**inductive-cases** AssignLocalsE[elim!]: $\pi:($Assign- Locals l,s) $\Rightarrow$ t
inductive-cases SeqE[elim!]: \( \pi: (c1;;c2,s1) \Rightarrow s3 \)

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

inductive-cases ScopeE[elim!]: \( \pi: (SCOPE c,s) \Rightarrow t \)

inductive-cases PCallE[elim!]: \( \pi: (PCall p,s) \Rightarrow t \)

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

inductive-cases WhileE[elim]: \( \pi: (WHILE b DO c,s) \Rightarrow t \)

2.4.2 Automatic Derivation

lemma Assign': \( s' = s(x := (s x)(aval i s := aval a s)) \Rightarrow \pi:x[i] := a, s) \Rightarrow s' \) by auto

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

lemma ArrayClear': \( s' = s(x := (\lambda-. 0)) \Rightarrow \pi:(CLEAR x[], s) \Rightarrow s' \) by auto

lemma Scope': \( s_1 = <<_s> \Rightarrow \pi:(c,s_1) \Rightarrow t \Rightarrow t' = <s|t> \Rightarrow \pi:(c,s,s_1) \Rightarrow t' \) by auto

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":=(\(\lambda\cdot 5\)>\) \(\Rightarrow\) \(\forall\)

2.5 Command Equivalence

Two commands are equivalent if they have the same semantics.

definition
equiv-c :: com \(\Rightarrow\) com \(\Rightarrow\) bool (infix \(\sim\) 50) where
c \(\sim\) c' \(\equiv\) (\(\forall\) s t. \(\pi\):(c,s) \(\Rightarrow\) t = \(\pi\):(c',s) \(\Rightarrow\) t)

lemma equivI[intro?]: 
\(\forall\) s t. \(\pi\):(c,s) \(\Rightarrow\) t \(\Rightarrow\) \(\pi\):(c',s) \(\Rightarrow\) t
\(\Rightarrow\) c \(\sim\) c'
by (auto simp: equiv-c-def)

lemma equivD[dest]: c \(\sim\) c' \(\Rightarrow\) \(\pi\):(c,s) \(\Rightarrow\) t \(\longleftrightarrow\) \(\pi\):(c',s) \(\Rightarrow\) t
by (auto simp: equiv-c-def)

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

lemma equiv-refl[simp, intro!]: c \(\sim\) c
by (blast intro: equivI)

lemma equiv-sym[sym]: (c \(\sim\) c') =\(\Rightarrow\) (c' \(\sim\) c)
by (blast intro: equivI)

lemma equiv-trans[trans]: c \(\sim\) c' \(\Rightarrow\) c' \(\sim\) c'' \(\Rightarrow\) c \(\sim\) c''
by (blast intro: equivI)

2.5.1 Basic Equivalences

lemma while-unfold:
(\WHILE b DO c) \(\sim\) (IF b THEN c ;; \WHILE b DO c ELSE SKIP)
by rule auto

lemma triv-if:
(IF b THEN c ELSE c) \(\sim\) c
by (auto intro!: equivI)

lemma commute-if:
(IF b THEN (IF b2 THEN c11 ELSE c12) ELSE c2) \(\sim\)
(IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12 ELSE c2))
by (auto intro!: equivI)

lemma sim-while-cong-aux:
[\pi; (\texttt{WHILE \ b \ DO \ c,s}) \Rightarrow \ t; \ bval \ b = bval \ b'; \ c \sim c'] \Longrightarrow \ \pi; (\texttt{WHILE \ b' \ DO \ c',s}) \Rightarrow \ t]

by (induction \ WHILE \ b \ DO \ c \ s \ t \ arbitrary; \ b \ c \ rule: \ big-step-induct)

auto

lemma sim-while-cong: \ bval \ b = bval \ b' \Rightarrow c \sim c' \Longrightarrow \ WHILE \ b \ DO \ c \sim \ WHILE \ b' \ DO \ c'

using equiv-c-def sim-while-cong-aux by auto

2.6 Execution is Deterministic

This proof is automatic.

theorem big-step-determ: \ [\pi; (c,s) \Rightarrow \ t; \ \pi; \ (c,s) \Rightarrow \ u] \Longrightarrow \ u = t

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

case (WhileTrue \ b \ s1 \ c \ s2 \ s3)

then show \ ?case \ by \ blast

qed fastforce+

2.7 Small-Step Semantics

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

Moreover, the step function is explicitly partial, returning \texttt{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 \texttt{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 \texttt{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 \texttt{SKIP}. Then, the whole command is reduced to \texttt{SKIP}.

fun small-step :: program \Rightarrow \ com \times \ state \rightarrow \ com \times \ state

small-step \pi \ (x[i]::=a,s) = Some \ (\texttt{SKIP}, \ s(x := (s \ x)(aval \ i \ s := \ aval \ a \ s)))

| small-step \ (x[i]::=y,s) = Some \ (\texttt{SKIP}, \ s(x := s \ y))
| small-step \ (\texttt{CLEAR} x[s],s) = Some \ (\texttt{SKIP}, \ s(x := (\lambda-, \ 0)))
| small-step \ (Assign-Locals l,s) = Some \ (\texttt{SKIP,<l|s>})
| small-step \ (\texttt{SKIP};c,s) = Some \ (c,s)
| small-step \( \pi \ (c_1; c_2, s) = (\text{case small-step} \ \pi \ (c_1, s) \ \text{of} \ Some \ (c_1', s')) \): \Rightarrow \ Some \ (c_1'; c_2, s') | - \Rightarrow \ None \\
| small-step \( \pi \ (\text{IF} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2, s) = \text{Some} \ (\text{if} \ \text{bool} \ b \ \text{then} \ (c_1, s) \ \text{else} \ (c_2, s)) \\
| small-step \( \pi \ (\text{SCOPE} \ c, s) = \text{Some} \ (c; \text{Assign-Locals} \ s, \text{<<}|s>|) \\
| small-step \( \pi \ (\text{WHILE} \ b \ \text{DO} \ c, s) = \text{Some} \ (\text{IF} \ b \ \text{THEN} \ c_1; \text{WHILE} \ b \ \text{DO} \ c \ \text{ELSE} \ \text{SKIP}, \ s) \\
| small-step \( \pi \ (\text{PCall} \ p, s) = (\text{case} \ \pi \ p \ \text{of} \ Some \ (c, s) \ | - \Rightarrow \ None) \\
| small-step \( \pi \ (\text{PScope} \ p' \ \text{SKIP}, s) = \text{Some} \ (\text{SKIP}, s) \\
| small-step \( \pi \ (\text{PScope} \ p' \ s, s') = (\text{case} \ \pi \ p' \ (c, s) \ \text{of} \ Some \ (c', s') \Rightarrow \ Some \ (\text{PScope} \ p' \ c', s') | - \Rightarrow \ None) \\
| small-step \( \pi \ (\text{SKIP}, s) = \text{Some} \ (\text{SKIP}, s) \\

We define the reflexive transitive closure of the step function.

\[
\text{inductive \ small-steps :: program \Rightarrow \ com \times state \Rightarrow (com \times state) }
\text{option \Rightarrow bool where }
\]

| simp: small-steps \( \pi \ cs \ (\text{Some} \ cs) \\
| \[ \text{small-step} \ \pi \ cs = \text{None} \] \Rightarrow \text{small-steps} \ \pi \ cs \ \text{None} \\
| \[ \text{small-step} \ \pi \ cs = \text{Some} \ cs1; \text{small-steps} \ \pi \ cs1 \ cs2 \] \Rightarrow \text{small-steps} \ \pi \ cs \ cs2 \\

lemma \ small-steps-append: small-steps \( \pi \ cs1 \ (\text{Some} \ cs2) \Rightarrow \text{small-steps} \ \pi \ cs2 \ cs3 \Rightarrow \text{small-steps} \ \pi \ cs1 \ cs3 \\
apply \ (\text{induction} \ \pi \ cs1 \ \text{Some} \ cs2 \ \text{arbitrary}; \ cs2 \ \text{rule: small-steps.induct}) \\
apply \ (\text{auto intro: small-steps.intros}) \\
done \\

2.7.1 Equivalence to Big-Step Semantics

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

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

lemma \ small-big-append: small-step \( \pi \ cs1 \ = \text{Some} \ cs2 \ \Rightarrow \pi; \ cs2 \Rightarrow \text{if} \ s3 \ \Rightarrow \pi; \ cs1 \Rightarrow \text{if} s3 \\
apply \ (\text{induction} \ \pi \ cs1 \ \text{arbitrary}; \ cs2 \ s3 \ \text{rule: small-step.induct}) \\
apply \ (\text{auto split: option.splits if-splits}) \\
done \\

lemma \ smalls-big-append: small-steps \( \pi \ cs1 \ (\text{Some} \ cs2) \Rightarrow \pi; \ cs2 \Rightarrow \text{if} \ s3 \ \Rightarrow \pi; \ cs1 \Rightarrow \text{if} s3 \\
apply \ (\text{induction} \ \pi \ cs1 \ \text{Some} \ cs2 \ \text{arbitrary}; \ cs2 \ \text{rule: small-steps.induct}) \\
apply \ (\text{auto intro: small-big-append}) \\
done \\

lemma \ small-imp-big:
assumes small-steps π cs₁ (Some (SKIP, s₂))
shows π: cs₁ ⇒ s₂
using smalls-big-append[OF assms]
by auto

lemma small-steps-skip-term [simp]:
  small-steps π (SKIP, s) cs' ←→
  cs' = Some (SKIP, s)
apply rule
subgoal
  apply (induction π (SKIP, s) cs' arbitrary: s rule: small-steps.induct)
  by (auto intro: small-steps.intros)
by (auto intro: small-steps.intros)

lemma small-seq: [c≠SKIP; small-step π (c, s) = Some (c', s')] ⇒
  small-step π (c; cx, s) = Some (c'; cx, s')
apply (induction π (c, s) arbitrary: c s c' s' rule: small-step.induct)
apply auto
done

lemma smalls-seq:
  [small-steps π (c, s) (Some (c', s'))] ⇒
  small-steps π (c; cx, s) (Some (c'; cx, s'))
apply (induction π (c, s) Some (c', s') arbitrary: c s c' s' rule: small-steps.induct)
apply (auto dest: small-seq intro: small-steps.intros)
by (metis option.simps(1) prod.simps(1) small-seq small-step.simps(31)
small-steps.intros(3))

lemma small-pscope:
  [c≠SKIP; small-step π' (c, s) = Some (c', s')] ⇒
  small-step π (PScope π' c, s) = Some (PScope π' c', s')
apply (induction π (c, s) arbitrary: c s c' s' rule: small-step.induct)
apply auto
done

lemma smalls-pscope:
  small-steps π' (c, s) (Some (c', s')) ⇒
  small-steps π (PScope π' c, s) (Some (PScope π' c', s'))
apply (induction π' (c, s) (Some (c', s')) arbitrary: c s rule: small-steps.induct)
apply auto
by (metis (no-types, hide-lams) option.inject prod.inject small-pscope
small-steps.simps small-steps-append small-steps-skip-term)

lemma big-imp-small:
assumes π: cs ⇒ t
shows small-steps π cs (Some (SKIP, t))
using assms
proof induction
case (Skip π s)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (AssignIdx π x i a s)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (ArrayCpy π x y s)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (ArrayClear π x s)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (Seq π c1 s1 s2 s3)
then show ?case by (meson small-step simp small-steps.intros simp small-steps-append smalls-seq)
next
case (IfTrue b s π c1 t c2)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (IfFalse b s π c2 t c1)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (Scope π c s s′)
then show ?case by (meson small-step simp small-step simp small-step simp small-step simp small-steps.intros simp small-steps.append smalls-seq)
next
case (WhileFalse b s π c)
then show ?case by (auto 0 4 intro: small-steps.intros)
next
case (WhileTrue b s1 π c s2 s3)
then show ?case proof
have ∀ ca p. (small-steps π p (Some (SKIP, s3)) ∨ Some (ca, s2) \neq Some (WHILE b DO c, s2)) ∨ small-step π p \neq Some (c; ca, s1)
by (metis (no-types) WhileTrue.IH(1) WhileTrue.IH(2) small-step simp small-steps.intros simp small-steps-append smalls-seq)
then have ∀ ca cb cc. (small-steps π (IF b THEN cc ELSE ca, s1)
(Skip, s3)) ∨ Some (cb, s2) \neq Some (WHILE b DO c, s2)) ∨ Some (cc, s1) \neq Some (c; cb, s1)
using WhileTrue.hyps(1) by force
then show ?thesis
using small-step simp small-steps.intros simp small-steps.intros simp small-steps.append smalls-seq by blast
qed
next
case (PCall π p c s t)
then show ?case by (auto 0 4 intro: small-steps.intros)
The big-step semantics yields a state \( t \), if and only if there is a transition of the small-step semantics to \((\text{SKIP}, t)\).

**Theorem big-eq-small**: \( \pi : \text{cs} \Rightarrow t \leftrightarrow \text{small-steps} \pi \text{cs} (\text{Some} (\text{SKIP}, t)) \)

**Lemma small-steps-determ**:

- **Assumes**: \( \text{small-steps} \pi \text{cs None} \)
- **Shows**: \( \neg \text{small-steps} \pi \text{cs} (\text{Some} (\text{SKIP}, t)) \)
- **Using**: \( \text{assms} \)
- **Apply**: \((\text{induction} \pi \text{cs None}::(\text{com} \times \text{state}) \text{option arbitrary}: t \text{ rule:}\text{small-steps.induct}) \)
- **Apply**: \((\text{auto elim: small-steps.cases}) \)
- **Done**

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

**Corollary small-imp-big-fail**:  
- **Assumes**: \( \text{small-steps} \pi \text{cs None} \)  
- **Shows**: \( \not\exists t. \pi : \text{cs} \Rightarrow t \)  
- **Using**: \( \text{assms} \)  
- **By**: \((\text{auto simp: big-eq-small small-steps-determ}) \)

### 2.8 Weakest Precondition

The following definitions are made wrt. a fixed program \( \pi \), 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**:  
\[ wp \ c \ Q \ s \equiv \exists t. \pi : (c, s) \Rightarrow t \land Q \ t \]

---

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** \( wlp \ c \ Q \ s \equiv \forall t. \pi;(c,s) \Rightarrow t \rightarrow Q t \)

### 2.8.1 Basic Properties

**context**

**notes** \([abs-def,simp]=wp-def wlp-def\)

**begin**

**lemma** \( wp\text{-}imp\text{-}wlp \): \( wp \ c \ Q \ s \Longrightarrow wlp \ c \ Q \ s \)

**using** \( big\text{-}step\text{-}determ \) \textbf{by force}

**lemma** \( wlp\text{-}and\text{-}term\text{-}imp\text{-}wp \): \( wlp \ c \ Q \ s \land \pi;(c,s) \Rightarrow t \Longrightarrow wp \ c \ Q \ s \)

**by** \( auto \)

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

**lemma** \( wp\text{-}conseq \): \( wp \ c \ P \ s \Longrightarrow [\[\forall s. P s \Rightarrow Q s\]] \Longrightarrow wp \ c \ Q \ s \)

**by** \( auto \)

**lemma** \( wp\text{-}equiv \): \( c \sim c' \Longrightarrow wlp \ c = wlp \ c' \) \textbf{by auto}

**lemma** \( wlp\text{-}conseq \): \( wlp \ c \ P \ s \Longrightarrow [\[\forall s. P s \Rightarrow Q s\]] \Longrightarrow wlp \ c \ Q \ s \)

**by** \( auto \)

### 2.8.2 Unfold Rules

**lemma** \( wp\text{-}skip\text{-}eq \): \( wp \ SKIP \ Q \ s = Q s \)

**by** \( auto \)

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

**by** \( auto \)

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

**by** \( auto \)

**lemma** \( wp\text{-}arrayinit\text{-}eq \): \( wp \ (CLEAR x][]) \ Q s = Q (s(x:=(λ-. 0))) \)

**by** \( auto \)

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

**by** \( auto \)

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

**by** \( auto \)

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

**by** \( auto \)

**lemma** \( wp\text{-}scope\text{-}eq \): \( wp \ (SCOPE c) \ Q s = wp \ c (λs'. Q <s|s'>) \)

**by** \( auto \)

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

**by** \( auto \)

**lemmas** \( wp\text{-}eq = wp\text{-}skip\text{-}eq wp\text{-}assign\text{-}idx\text{-}eq wp\text{-}arraycpy\text{-}eq wp\text{-}arrayinit\text{-}eq \)

**lemmas** \( wp\text{-}assign\text{-}locals\text{-}eq wp\text{-}seq\text{-}eq wp\text{-}scope\text{-}eq \)

**lemmas** \( wp\text{-}eq' \ = \ wp\text{-}eq wp\text{-}if\text{-}eq \)

**lemma** \( wlp\text{-}skip\text{-}eq \): \( wlp \ SKIP \ Q \ s = Q s \)

**by** \( auto \)
lemma wlp-assign-idx-eq: wlp (x[i]:=a) Q s = Q (s(x:=(s x)(aval i s := aval a s))) by auto
lemma wlp-arraycpy-eq: wlp (x[][]:=a) Q s = Q (s(x:=s a)) by auto
lemma wlp-arrayinit-eq: wlp (CLEAR x[]) Q s = Q (s(x:=(λ.. 0))) by auto
lemma wlp-assign-locals-eq: wlp (Assign-Locals l) Q s = Q <l|s> by auto
lemma wlp-seq-eq: wlp (c1;;c2) Q s = wlp c1 (wlp c2 Q s) by auto
lemma wlp-if-eq: wlp (IF b THEN c1 ELSE c2) Q s = (if bval b s then wlp c1 Q s else wlp c2 Q s) by auto
lemma wlp-scope-eq: wlp (SCOPE c) Q s = wlp c (λs′. Q <s|s'>) by auto
lemma wlp-pcall-eq: π p = Some c =⇒ wlp (PCall p) Q s = wlp c Q s by auto

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

lemma wlp-while-unfold: wlp (WHILE b DO c) Q s = (if bval b s then wlp c (wlp (WHILE b DO c) Q s)) s else Q s)
  apply (subst wlp-equiv[OF while-unfold])
  apply (simp add: wlp-eq′)
done

lemma wp-while-unfold: wp (WHILE b DO c) Q s = (if bval b s then wp c (wp (WHILE b DO c) Q s)) s else Q s)
  apply (subst wp-equiv[OF while-unfold])
  apply (simp add: wp-eq′)
done
end

— Context fixing program
Unfold rules for procedure scope
lemma wp-pscope-eq: wp (PScope π' c) Q s = wp π' (c) Q s
  unfolding wp-def by auto
lemma wp-pscope-eq: wlp π (PScope π' c) Q s = wlp π' (c) Q s
  unfolding wlp-def by auto

2.8.3 Weakest precondition and Program Equivalence

The following three statements are equivalent:
1. The commands \( c \) and \( c' \) are equivalent

2. The weakest preconditions are equivalent, for all procedure environments

3. The weakest liberal preconditions are equivalent, for all procedure environments

\textbf{lemma \( \text{wp-equiv-iff} \):} \( \forall \pi. \text{wp} \pi c = \text{wp} \pi c' \) \( \iff \) \( c \sim c' \)

\textbf{unfolding} \( \text{equiv-c-def} \) \text{using} \( \text{big-step-determ} \) \text{unfolding} \( \text{wp-def} \) \text{by} (auto; metis)

\textbf{lemma \( \text{wlp-equiv-iff} \):} \( \forall \pi. \text{wlp} \pi c = \text{wlp} \pi c' \) \( \iff \) \( c \sim c' \)

\textbf{unfolding} \( \text{equiv-c-def wlp-def} \) \text{by} (auto; metis (no-types, hide-lams))

\subsection*{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.

\textbf{lemma \( \text{sim-while-wp-aux} \):}

\begin{itemize}
\item assumes \( \text{bval \( b = \) bval \( b' \)} \)
\item assumes \( \text{wp} \pi c = \text{wp} \pi c' \)
\item assumes \( \pi: (\text{WHILE \( b \) DO \( c \)}, s) \Rightarrow t \)
\item shows \( \pi: (\text{WHILE \( b' \) DO \( c' \)}, s) \Rightarrow t \)
\item using \( \text{assms(3,2)} \)
\item apply (induction \( \pi \) \text{WHILE \( b \) DO \( c \) \( s \ t \)})
\item apply (auto simp: \( \text{assms(1)} \))
\item by (metis WhileTrue \text{big-step-determ} \text{wp-def})
\end{itemize}

\textbf{lemma \( \text{sim-while-wp} \):} \( \text{bval \( b = \) bval \( b' \)} \Rightarrow \text{wp} \pi c = \text{wp} \pi c' \Rightarrow \text{wp} \pi (\text{WHILE \( b \) DO \( c \})) = \text{wp} \pi (\text{WHILE \( b' \) DO \( c' \)}) \)

\begin{itemize}
\item apply (intro ext)
\item apply (auto 0 3 simp: \text{wp-def intro: \( \text{sim-while-wp-aux} \)})
\item done
\end{itemize}

The same lemma for weakest liberal preconditions.

\textbf{lemma \( \text{sim-while-wlp-aux} \):}

\begin{itemize}
\item assumes \( \text{bval \( b = \) bval \( b' \)} \)
\item assumes \( \text{wlp} \pi c = \text{wlp} \pi c' \)
\item assumes \( \pi: (\text{WHILE \( b \) DO \( c \)}, s) \Rightarrow t \)
\item shows \( \pi: (\text{WHILE \( b' \) DO \( c' \)}, s) \Rightarrow t \)
\item using \( \text{assms(3,2)} \)
\item apply (induction \( \pi \) \text{WHILE \( b \) DO \( c \) \( s \ t \)})
\item apply (auto simp: \( \text{assms(1,2)} \))
\item by (metis WhileTrue \text{wp-def})
\end{itemize}
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 s₀
  assumes STEP: \( \forall s . I s \implies (\text{if } \text{bval } b s \text{ then } \text{wlp } \pi c I s \text{ else } Q s) \)
  shows \( \text{wlp } \pi (\text{WHILE } b \text{ DO } c) Q s₀ \)
unfolding wlp-def
proof clarify
fix t
assume \( \pi : (\text{WHILE } b \text{ DO } c, s₀) \implies t \)
thus \( Q t \) using INIT STEP
proof (induction \( \pi \) WHILE b DO c s₀ t rule: big-step-induct)
case (WhileFalse s) with STEP show Q s by auto
next
case (WhileTrue s₁ π s₂ s₃)
  note STEP' = WhileTrue.prems(2)
  from STEP'[OF (I s₁) (bval b s₁)] have \( \text{wlp } \pi c I s₁ \) by simp
  with \( \pi : (c, s₁) \implies s₂ \) have \( I s₂ \) unfolding wlp-def by blast
  moreover have \( I s₂ \implies Q s₃ \) using STEP' WhileTrue.hyps(5)
by blast
  ultimately show Q s₃ by blast
qed
qed
```

```
lemma
  assumes INIT: I s₀
  assumes STEP: \( \forall s . I s \implies (\text{if } \text{bval } b s \text{ then } \text{wlp } \pi c I s \text{ else } Q s) \)
  shows \( \text{wlp } \pi (\text{WHILE } b \text{ DO } c) Q s₀ \)
  using STEP
unfolding wlp-def
apply clarify subgoal premises prems for t
  using prems(2,1) INIT
by (induction \( \pi \) WHILE b DO c s₀ t rule: big-step-induct; meson)
done
```
2.9.2 Total Correctness

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

**Lemma wp-whileI**:  
assumes \(WF: wf R\)  
assumes \(INIT: I s_0\)  
assumes \(STEP: \forall s. I s \Rightarrow (if bval b s then wp \pi c (\lambda s'. I s' \land (s',s)\in R) \ s \ else \ Q s)\)  
shows \(wp \pi (WHILE b DO c) \ Q s_0\)  
using \(WF \ INIT\)

**Proof** (induction rule: \(wf-induct-rule[where a=s_0]\))

- \(case (less s)\)
  - \(show \ wp \pi (WHILE b DO c) \ Q s\)
  - \(proof (rule wp-while-unfold[THEN ifD2])\)
    - \(show if bval b s \ then \ wp \pi c (wp \pi (WHILE b DO c) \ Q) \ s \ else \ Q s\)
      - \(proof (split if-split; intro allI impI conjI)\)
        - \(assume [simp]: bval b s\)
          - \(from \ STEP (I s) \ have \ wp \pi c (\lambda s'. I s' \land (s',s)\in R) \ s \ by \ simp\)
            - \(thus \ wp \pi c (wp \pi (WHILE b DO c) \ Q) \ s \ proof (rule wp-conseq)\)
              - \(fix s' assume I s' \land (s',s)\in R\)
                - \(with \ less.IH \ show \ wp \pi (WHILE b DO c) \ Q s' \ by \ blast\)
          qed
          next
          - \(assume [simp]: \neg bval b s\)
            - \(from \ STEP (I s) \ show \ Q s \ by \ simp\)
          qed
          qed

**Lemma**  
assumes \(WF: wf R\)  
assumes \(INIT: I s_0\)  
assumes \(STEP: \forall s. I s \Rightarrow (if bval b s then wp \pi c (\lambda s'. I s' \land (s',s)\in R) \ s \ else \ Q s)\)  
shows \(wp \pi (WHILE b DO c) \ Q s_0\)  
using \(WF \ INIT\)

**Apply (induction rule: \(wf-induct-rule[where a=s_0]\))**

**Apply (subst wp-while-unfold)**

**By \(smt \ STEP \ wp-conseq)\)

2.9.3 Standard Forms of While Rules

**Lemma wlp-whileI**:  
assumes \(INIT: I s_0\)
assumes STEP: $I s; \text{val } b s \Rightarrow \text{wp } \pi (c) I s$

assumes FINAL: $I s; \neg \text{val } b s \Rightarrow Q s$

shows $\text{wp } \pi (\text{WHILE } b \text{ DO } c) Q s_0$

using assms wp-whileI' by auto

2.10 Modularity of Programs

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

lemma map-leD:
  assumes $m \subseteq m'$
  assumes $m x = \text{Some } v = m' x = \text{Some } v$
  by (metis domI map-le-def)

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

Wrapping a set of recursive procedures into a procedure scope

lemma localize-recursion:
  $\pi': (\text{PScope } \pi c, s) \Rightarrow t \longleftrightarrow \pi'(c, s) \Rightarrow t$
  by auto

2.11 Strongest Postcondition

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

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

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

Intuition: There exists an old value $v x$ for the assigned variable

lemma $\text{sp-arraycpy-eq}: sp P (x[]:=y) t \longleftrightarrow (\exists v x. \text{let } s = t(x:=v x) \in t x = s y \wedge P s)$
  apply (auto simp: big-step-simps)
  apply (intro cxi conjI, assumption, auto) []
  apply (intro cxi conjI, assumption, auto) []
Version with renaming of assigned variable

```ml
lemma sp-arraycpy-eq': sp P (x:::=y) t \iff t x = t y \land (\exists vx. P (t(x:=vx,y:=t x)))
apply (auto simp: big-step-simps)
apply (metis fun-upd-triv)
apply (intro exI conjI, assumption)
apply auto
done
```

```
lemma sp-skip-eq: sp P SKIP t \iff P t by auto
lemma sp-seq-eq: sp P (c1;;c2) t \iff sp (sp P c1) c2 t by auto
```

\end

2.12 Hoare-Triples

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

```ml
definition HT where HT π P c Q \equiv (\forall s_0. P s_0 \rightarrow wp \pi c (Q s_0) s_0)
definition HT-partial where HT-partial π P c Q \equiv (\forall s_0. P s_0 \rightarrow wlp \pi c (Q s_0) s_0)
```

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

```ml
lemma HT-conseq:
assumes HT π P c Q
assumes s P' \equiv P .
assumes s_0 . [P s_0; P' s_0; Q s_0 s \equiv] \rightarrow Q' s_0 s
shows HT π P' c Q'
using assms unfolding HT-def by (blast intro: wp-conseq)
```

```ml
lemma HT-partial-conseq:
assumes HT-partial π P c Q
assumes s P' \equiv P s
assumes s_0 . [P s_0; P' s_0; Q s_0 s \equiv] \rightarrow Q' s_0 s
shows HT-partial π P' c Q'
using assms unfolding HT-partial-def by (blast intro: wlp-conseq)
```

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

```ml
lemma wp-modularity-rule:
[P\pi P c Q; P s; (\lambda s'. Q s s' \equiv Q' s')] \equiv wp \pi c Q' s
```

23
unfolding HT-def by (blast intro: wp-conseq)

2.12.1 Sets of Hoare-Triples

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

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

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

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

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

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

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

Intuition: Scope can be seen as assignment of locals before and after inner command. Thus, this rule is a combined forward and
backward assignment rule, for the epilogue locals:=<> and the prologue locals:=old-locals.

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

2.12.3 Proof for Recursive Specifications

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

lemma HTsetI:
assumes $wf R$
assumes $RL: \bigwedge P c Q s_0. \lbrack \lbrack HTset-r (\lambda c' s'. ((c',s'),(c,s_0))\in R ) \pi \Theta; (P,c,Q)\in \Theta; P s_0 \rbrack \rbrack wp \pi c (Q s_0) s_0$
shows $HTset \pi \Theta$
unfolding HTset-def HT-def
proof clarsimp
fix $P c Q s_0$
assume $(P,c,Q)\in \Theta P s_0$
with $wf R$ show $wp \pi c (Q s_0) s_0$
apply (induction $(c,s_0)$ arbitrary: $c s_0 P Q$)
using $RL$ unfolding HTset-r-def HT-def
by blast
qed

lemma HT-simple-recursiveI:
assumes $wf R$
assumes $\bigwedge s. \lbrack HT \pi (\lambda s'. (f s', f s)\in R \land P s') c Q; P s \rbrack \rbrack wp \pi c (Q s) s$
shows $HT \pi P c Q$
using HTsetI|where $R=\text{inv-image } R (f o \text{snd})$ and $\pi=\pi$ and $\Theta = \{(P,c,Q)\}$] assms
by (auto simp: HTset-r-def HTset-def)

lemma HT-simple-recursive-procI:
assumes $wf R$
assumes $\bigwedge s. \lbrack HT \pi (\lambda s'. (f s', f s)\in R \land P s') (P\text{Call } p) Q; P s \rbrack \rbrack wp \pi (P\text{Call } p) (Q s) s$
shows $HT \pi P (P\text{Call } p) Q$
using HTsetI|where $R=\text{inv-image } R (f o \text{snd})$ and $\pi=\pi$ and $\Theta =$
\{(P,PCall p,Q)\} \text{ axsms}
by \text{(auto simp: HTset-r-def HTset-def)}

\begin{itemize}
\item \text{lemma}
\item \text{assumes} \(wf\ R\)
\item \text{assumes} \(\langle s P p Q \rangle \in \Theta\rightarrow HT\ \pi\ \{(\lambda s'. ((p',s'),(p,s))\in R \land P' s') (PCall p') Q'\}
(P,p,Q)\in \Theta; P s \rangle \rightarrow wp\ \pi\ (PCall p) (Q s) s\)
\item \text{shows} \(\forall (P,p,Q)\in \Theta.\ HT\ \pi\ P\ (PCall p) Q\)
\item \text{proof --}
\item \text{have} \(HTset\ \pi\ \{(P,PCall p,Q) | P p Q. (P,p,Q)\in \Theta\}\)
\item \text{apply (rule HTsetI[where \(R=\text{ine-image}\ R (\lambda x.\ \text{case}\ x\ of\ (PCall p,s)\Rightarrow (p,s))])\}
\item \text{subgoal using} \(\langle wf\ R:\ by\ simp\)
\item \text{subgoal for} \(P c Q s\)
\item \text{apply clarsimp}
\item \text{apply (rule axsms(2)[where \(P=P]\))}
\item \text{apply simp-all}
\item \text{unfolding HTset-r-def}
\item \text{proof goal-cases}
\item \text{case} \((1\ p\ P'\ p'\ Q')\)
\item \text{from} \(1(1)[\text{rule-format, of} \ (P',PCall p',Q'),\ \text{simplified}]\)
\item \text{show} \(\text{?case by auto}\)
\item \text{qed}
\item \text{done}
\item \text{thus} \(\text{?thesis by (auto simp: HTset-def)}\)
\item \text{qed}
\end{itemize}

2.13 Completeness of While-Rule

\begin{itemize}
\item Idea: Use \(wlp\) as invariant
\item \text{lemma} \(wlp\text{-while}\ I\text{-complete:}\)
\item \text{assumes} \(wlp\ \pi\ (\text{WHILE}\ b\ DO\ c)\ Q\ s_0\)
\item \text{obtains} \(I\ where\)
\item \(I\ s_0\)
\item \(\langle s.\ I\ s\rightarrow \text{if bval}\ b\ s\ \text{then}\ wlp\ \pi\ c\ I\ s\ \text{else}\ Q\ s\\)
\item \text{proof}
\item \text{let} \(\text{?I} = wlp\ \pi\ (\text{WHILE}\ b\ DO\ c)\ Q\)
\item \{\text{show} \(\text{?I}\ s_0\ \text{by}\ \text{fact}\}
\item \text{next}
\item \text{fix} \(s\)
\end{itemize}
assume \( I \ s \)
then show if \( \text{bval} \ b \ s \) then \( \text{wlp} \ \pi \ c \ I \ s \) else \( Q \ s \)
apply (subst (asm) \( \text{wlp-while-unfold} \))
.
}

qed

Idea: Remaining loop iterations as variant

\textbf{inductive} count-it for \( \pi \ b \ c \) where
\[
\neg \text{bval} \ b \ s \implies \text{count-it} \ \pi \ b \ c \ s \ 0
| \ [ \text{bval} \ b \ s ; \ \pi ; (c,s) \Rightarrow s' ; \text{count-it} \ \pi \ b \ c \ s' \ n ] \implies \text{count-it} \ \pi \ b \ c \ s (\text{Suc} \ n)
\]

\textbf{lemma} count-it-determ:
\[
\text{count-it} \ \pi \ b \ c \ s \ n \implies \text{count-it} \ \pi \ b \ c \ s \ n' \implies n' = n
\]
apply (induction arbitrary: \( n' \) rule: count-it.induct)
subgoal using count-it.cases by blast
subgoal by (metis \( \text{big-step-determ} \) count-it.cases)
done

\textbf{lemma} count-it-ex:
assumes \( \pi; (\text{WHILE} \ b \ DO \ c,s) \Rightarrow t \)
shows \( \exists n. \text{count-it} \ \pi \ b \ c \ s \ n \)
using assms
apply (induction \( \pi \) \text{WHILE} \( b \) DO \( c \), \( s \) arbitrary: \( b \) \( c \))
apply (auto intro: count-it.intros)
done

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

\textbf{lemma} variant-decreases:
assumes \( \text{STEPB} \): \( \text{bval} \ b \ s \)
assumes \( \text{STEPC} \): \( \pi; (c,s) \Rightarrow s' \)
assumes \( \text{TERM} \): \( \pi; (\text{WHILE} \ b \ DO \ c,s') \Rightarrow t \)
shows \( \text{variant} \ \pi \ b \ c \ s' < \text{variant} \ \pi \ b \ c \ s \)

\textbf{proof} –
from count-it-ex[OF TERM] obtain \( n' \) where \( CI' \): \( \text{count-it} \ \pi \ b \ c \ s' \ n' \)
moreover from count-it.intros(2)[OF STEPB STEPC this] have count-it \( \pi \ b \ c \ s \ (\text{Suc} \ n') \).
ultimately have \( \text{variant} \ \pi \ b \ c \ s' = n' \) \( \text{variant} \ \pi \ b \ c \ s = \text{Suc} \ n' \)
unfolding variant-def using count-it-determ by blast+
thus ?thesis by simp
qed

\textbf{lemma} \( \text{wp-whileI}'\)-complete:
fixes \( \pi \ b \ c \)
defines \( R \equiv \text{measure} \ (\text{variant} \ \pi \ b \ c) \)
assumes \( \text{wp} \ \pi; (\text{WHILE} \ b \ DO \ c) \ Q \ s_0 \)
obtains \( \langle \omega f R \rangle \) where
\[
\omega f R \quad I s_0 \quad \forall s. I s \implies \text{if } b \text{val } b s \text{ then } \omega f \pi c (\lambda s'. I s' \land (s',s) \in R) s \text{ else } Q s
\]

proof
\begin{align*}
&\text{show } \langle \omega f R \rangle \text{ unfolding } R\text{-def by auto} \\
&\text{let } ?I = \omega f \pi (\text{WHILE } b \text{ DO } c) Q \\
&\begin{cases}
&\text{show } ?I s_0 \text{ by fact} \\
&\text{next} \\
&\text{fix } s \\
&\text{assume } ?I s \\
&\text{then show if } b \text{val } b s \text{ then } \omega f \pi c (\lambda s'. ?I s' \land (s',s) \in R) s \text{ else } Q s
\end{cases} \\
&\text{apply } (\text{subst } (\text{asm}) \omega f\text{-while-unfold}) \\
&\text{apply } \text{clarsimp} \\
&\text{by } (\text{auto simp: } \omega f\text{-def } R\text{-def intro: } \text{variant-decreases})
\end{align*}

qed

end

3 Annotated Syntax

theory Annotated-Syntax
imports Semantics
begin

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

named-theorems vcg-annotation-defs (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 :: \((\text{state } \Rightarrow \text{bool}) \Rightarrow \text{bexp} \Rightarrow \text{com} \Rightarrow \text{com}\)
WHILE \{ \} \{- \} \{- \} \{- \} \{- \} DO - \\
where \{vcg-annotation-defs\}: WHILE-annotI \( I::\text{state} \Rightarrow \text{bool} \) \equiv \text{While}

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

definition WHILE-annotRVI :: 'a rel \Rightarrow (\text{state} \Rightarrow 'a) \Rightarrow (\text{state} \Rightarrow \text{bool}) \Rightarrow \text{bexp} \Rightarrow \text{com} \Rightarrow \text{com} \\
\begin{align*}
\begin{array}{ll}
((\text{WHILE } \{ \} \{- \} \{- \} \{- \} \{- \} DO - ) & [0, 0, 0, 0, 61] 61) \\
\text{where} \{vcg-annotation-defs\}: \text{WHILE-annotRVI } R \text{ } V \text{ } I \equiv \text{While for } R \text{ } V \text{ } I
\end{array}
\end{align*}

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

definition WHILE-annotVI :: (\text{state} \Rightarrow \text{int}) \Rightarrow (\text{state} \Rightarrow \text{bool}) \Rightarrow \text{bexp} \Rightarrow \text{com} \Rightarrow \text{com} \\
\begin{align*}
\begin{array}{ll}
((\text{WHILE } \{ \} \{- \} \{- \} \{- \} \{- \} DO - ) & [0, 0, 0, 0, 61] 61) \\
\text{where} \{vcg-annotation-defs\}: \text{WHILE-annotVI } V \text{ } I \equiv \text{While for } V \text{ } I
\end{array}
\end{align*}

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!

\textbf{type-synonym} \( HT'\text{-type} \equiv \text{program} \Rightarrow (\text{state} \Rightarrow \text{bool}) \Rightarrow (\text{state} \Rightarrow \text{com}) \Rightarrow (\text{state} \Rightarrow \text{state} \Rightarrow \text{bool}) \Rightarrow \text{bool} \)

definition \(HT'\text{-partial} \equiv HT'\text{-type}
\begin{align*}
\begin{array}{ll}
\text{where} & HT'\text{-partial} \pi \text{ } P \text{ } c \text{ } Q \equiv \forall \text{ } s_0. \text{ } P \text{ } s_0 \longrightarrow \text{wp} \pi \text{ (c } s_0) \text{ (Q } s_0) \text{ s}_0
\end{array}
\end{align*}

definition \(HT' \equiv HT'\text{-type}
\begin{align*}
\begin{array}{ll}
\text{where} & HT' \pi \text{ } P \text{ } c \text{ } Q \equiv \forall \text{ } s_0. \text{ } P \text{ } s_0 \longrightarrow \text{wp} \pi \text{ (c } s_0) \text{ (Q } s_0) \text{ s}_0
\end{array}
\end{align*}

\textbf{lemma} \( HT'\text{-eq-HT: } HT' \pi \text{ } P \text{ (\lambda- c) } Q \equiv HT \pi \text{ } P \text{ } c \text{ } Q \)
\textbf{unfolding} \( HT\text{-def } HT'\text{-def} \text{ ..} \)

\textbf{lemma} \( HT'\text{-partial-eq-HT: } HT'\text{-partial} \pi \text{ } P \text{ (\lambda- c) } Q \equiv HT'\text{-partial} \pi \text{ } P \text{ } c \text{ } Q \)
\textbf{unfolding} \( HT'\text{-partial-def } HT'\text{-partial-def} \text{ ..} \)

\textbf{lemmas} \( HT'\text{-anfolds } = HT'\text{-eq-HT } HT'\text{-partial-eq-HT} \)

type-synonym \( 'a \Theta \text{elem-t } = (\text{state} \Rightarrow 'a) \times ((\text{state} \Rightarrow \text{bool}) \times (\text{state} \Rightarrow \text{state} \Rightarrow \text{bool})) \)
definition \( \text{HT}'\text{set} :: \text{program} \Rightarrow 'a \Theta \text{elem-t set} \Rightarrow \text{bool} \) where  
\[ \text{HT}'\text{set} \pi \Theta \equiv \forall (n, (P, c, Q)) \in \Theta. \text{HT}' \pi P c Q \]

definition \( \text{HT}'\text{set-r} :: 'a \Rightarrow \text{program} \Rightarrow 'a \Theta \text{elem-t set} \Rightarrow \text{bool} \) where  
\[ \text{HT}'\text{set-r} r \pi \Theta \equiv \forall (n, (P, c, Q)) \in \Theta. \text{HT}' \pi (\lambda s. r n s \wedge P s) c Q \]

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

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

end

4 Quickstart Guide

theory Quickstart-Guide
imports ../IMP2
begin

4.1 Introductory Examples

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

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

The specification consists of the signature (name, parameters, return variables), precondition, postcondition, and program text.

\textbf{Signature} The procedure name and variable names must be valid Isabelle names. The \textit{returns} declaration is optional, by default, nothing is returned. Multiple values can be returned by \textit{returns} \langle x_1, \ldots, x_n \rangle.

\textbf{Precondition} An Isabelle formula. Parameter names are valid variables.

\textbf{Postcondition} An Isabelle formula over the return variables, and parameter names suffixed with \textit{o}.

\textbf{Program Text} The procedure body, in a C-like syntax.

The \textit{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 \textsc{IMP2}'s verification condition generator, followed by manual discharging of the generated VCs as necessary. Note that the \textsc{vcg-cs} method will apply \textit{clarsimp} to all generated VCs, which, in our case, already solves them. You can use \textit{vcg} to get the raw VCs.

If the VCs have been discharged, \textit{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.

\texttt{thm \textit{div-ab-spec} — Final theorem proved}
\texttt{thm \textit{div-ab-def} — Constant definition, with parameter passing code}

The final theorem has the form \textit{HT-mods} $\pi \ vs \ P \ c \ Q$, where $\pi$ is an arbitrary procedure environment, \textit{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 $\textit{state \Rightarrow bool}$. It starts with a series of variable bindings that map program variables to logical variables, followed by precondition that was specified, wrapped in a \textit{BB-PROTECT} constant, which serves as a tag for the VCG, and is defined as the identity ($\textit{BB-PROTECT} \equiv \lambda a. \ a$).
The final theorem is declared to the VCG, such that the specification will be used automatically for calls to this procedure.

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

### 4.1.1 Variant and Invariant Annotations

Loops must be annotated with variants and invariants.

\textbf{procedure-spec} \texttt{mult-ab(a,b)} \textbf{returns} \texttt{c} \textbf{assumes} \langle \text{True} \rangle \textbf{ensures} \langle c = a_0 \times b_0 \rangle \textbf{defines} \langle \begin{align*}
\text{if} \ (a < 0) \ & \begin{cases}
\ a = -a; & b = -b; \\
\ c = 0;
\end{cases} \\
\text{while} \ (a > 0) \\
\ @\text{variant} \langle a \rangle \ \\
\ @\text{invariant} \langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) \times b_0 \times \text{sgn} \ a_0 \rangle \ \\
\ \begin{cases}
\ c = c + b; \\
\ a = a - 1
\end{cases}
\end{align*}\rangle \textbf{apply} \texttt{vcg.cs}

The variant and invariant can use the program variables. Variables suffixed with \texttt{0} refer to the values of parameters at the start of the program.

The variant must be an expression of type \texttt{int}, which decreases with every loop iteration and is always $\geq 0$.

\textbf{Pitfall}: If the variant has a more general type, e.g., \texttt{'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 \texttt{rec} keyword.
recursive-spec
    odd-imp(n) returns b assumes n≥0 ensures \(b \neq 0 \iff \text{odd } n\)
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 \(b \neq 0 \iff \text{even } n\)
variant (n)
    defines (if \(n = 0\) b=1 else b=rec odd-imp(n-1))
by vcg-cs

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

thm odd-imp-spec even-imp-spec

4.2 The VCG

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

procedure-spec mult-ab(a,b) returns c assumes (True) ensures c=a0*b0
defines (if \(a<0\) \{a = -a; b = -b\};
c=0;
while (a>0)
  @variant (a)
  @invariant \(0 \leq a \land a \leq |a_0| \land c = (|a_0| - a) * b_0 * \text{sgn } a_0\)
  \{c=c+b;
a=a-1\}
)
apply vcg

The \(\cdot \)xxx tags in the premises give a hint to the origin of each VC. Moreover, the variable names are derived from the actual variable names in the program.
by (auto simp: algebra-simps)

4.3 Advanced Features

4.3.1 Custom Termination Relations

Both for loops and recursive procedures, a custom termination relation can be specified, with the relation annotation. The variant must be a function into the domain of this relation.
Pitfall: You have to ensure, by type annotations, that the most general type of the relation and variant fit together. Otherwise, ugly low-level errors will be the result.

```
procedure-spec `mult-ab` returns c assumes `(True)` ensures `c = a0 * b0`
defines (
  if `(a < 0)` `{ a = -a; b = -b };`
c = 0;
while `(a > 0)`
  @relation `(measure nat)`
  @variant `(a)`
  @invariant `(0 <= a ∧ a <= |a0|) ∧ c = (|a0| - a) * b0 * sgn a0`
  {
    c = c + b;
    a = a - 1
  }
) by vcg-cs (auto simp: algebra-simps)
```

```
recursive-spec relation `(measure nat)`
`odd-imp` 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` returns b assumes `n >= 0` ensures `(b ≠ 0 ←→ even n0)`
variant `(n)`
defines (if `(n = 0)` `b = 1` else `b = rec odd-imp `(n - 1)`);
by vcg-cs
```

4.3.2 Partial Correctness

IMP2 supports partial correctness proofs only for while-loops. Recursive procedures must always be proved totally correct\(^1\)

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

4.3.3 Arrays

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

\(^1\)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.
In the specifications, arrays are modeled as functions of type \( \text{int} \Rightarrow \text{int} \).

**lemma** array-sum-aux: \( l_0 \leq l \implies \{ l_0..<l + 1 \} = \text{insert} \ l \ \{ l_0..<l \} \)

**for** \( l_0 \ l :: \text{int} \) by auto

**procedure-spec** array-sum\((a,l,h)\) returns \( s \) assumes \( l\leq h \) ensures \( (s = \sum_{i=l_0..<h_0. \ a \ i}) \) defines

\( s=0; \)

while \( (l<h) \)

@variant \( h-l \)

@invariant \( l_0\leq l \land l \leq h \land s = (\sum_{i=l_0..<h_0. \ a \ i}) \)

\{ \ s = s+a[l]; \ l=l+1 \ \}\)

apply vcg-cs

apply (simp add: array-sum-aux)

done

### 4.4 Proving Techniques

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

#### 4.4.1 Auxiliary Lemmas

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

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

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

#### 4.4.2 Inlining

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

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

4.4.4 Data Refinement

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

4.5 Troubleshooting

We list a few common problems and their solutions here

4.5.1 Invalid Variables in Annotations

Undeclared variables in annotations are highlighted, however, no warning or error is produced. Usually, the generated VCs will not be provable. The most common mistake is to forget the 0 suffix when referring to parameter values in (in)variants and postconditions.

Note the highlighting of unused variables in the following example

```plaintext
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
    }
  }

oops
```

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

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

For total correctness, you must annotate a loop variant and invariant. For partial correctness, you must annotate an invariant, but no variant.
When not following this rule, the VCG will get stuck in an internal state

\[
\text{procedure-spec (partial) } \text{foo () assumes True ensures True defines}
\]
\[
\text{while } (n>0) \text{ variant } (n) \text{ invariant } (True)
\]
\[
\{ \text{n} = n - 1 \}
\]
\[
\text{apply vcg}
\]
\[
\text{oops}
\]

4.5.3 Calls to Undefined Procedures

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

4.6 Missing Features

This is an (incomplete) list of missing features.

4.6.1 Elaborate Warnings and Errors

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

4.6.2 Static Type Checking

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

Every variable is an integer array. 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.

\[
\text{datatype } \text{val} = \text{PRIM int} \mid \text{STRUCT } \text{fname} \Rightarrow \text{val} \mid \text{ARRAY int} \Rightarrow \text{val}
\]

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

\[
\begin{align*}
\textit{definition} \text{exp-count-up1} & \equiv \\
"a" & ::= N 1 ; \\
"c" & ::= N 0 ; \\
\text{WHILE} \text{Cmpop (<)} (V "c") (V "n") \text{DO} ( \\
"a" & ::= \text{Binop} (\ast) (N 2) (V "a"); \\
"c" & ::= \text{Binop} (+) (V "c") (N 1))
\end{align*}
\]

Fancy Syntax

\[
\begin{align*}
\textit{definition} \text{exp-count-up2} & \equiv \text{imp} \langle \\
- \text{Initialization} \\
a & = 1; \\
c & = 0; \\
\text{while} (c<n) \{ - \text{Iterate until } c \text{ has reached } n \\
a & = 2 \ast a; - \text{Double } a \\
c & = c + 1 - \text{Increment } c \\
\}
\end{align*}
\]

\textit{lemma} \text{exp-count-up1} = \text{exp-count-up2} (\text{unfolding} \text{exp-count-up1-def} \text{exp-count-up2-def} ..)
5.2 Operators and Arrays

We reflect arbitrary Isabelle functions into the syntax:

\[
\text{value } \text{bval} (\text{Cmpop } \leq) (\text{Binop } +) (\text{Unop } \text{uminus } (V \ ''x'')) (N 42) (N 50)) <'x'\text{:=}(\lambda\cdot -5)>
\]

\text{thm } \text{aval.simps bval.simps}

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

\[
\text{term } \langle V\text{idx} a (i::aexp) \rangle — \text{Array access at index } i
\]

\[
\text{lemma } V''x'' = V\text{idx} ''x'' (N 0) .. — \text{Shortcut for access at index 0}
\]

New commands:

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

\[
\text{term } (''a''[i] ::= v) — \text{Standard syntax}
\]

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

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

\[
\text{term } (''x'' ::= v) \text{ term } \langle \text{imp} x = v+1 \rangle
\]

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.

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

\[
\text{term } (''d''[] ::= ''s''[]) \text{ term } \langle \text{imp} d = s \rangle
\]

\[
\text{term } \langle \text{ArrayClear } ''a'' \rangle — \text{Initialize array to all zeroes.}
\]

\[
\text{term } \langle \text{CLEAR } ''a''[] \rangle \text{ term } \langle \text{imp} \text{clear } a[] \rangle
\]

Semantics of these is straightforward

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

5.3 Local and Global Variables

\[
\text{term } \langle \text{is-global} \rangle \text{ term } \langle \text{is-local} \rangle — \text{Partitions variable names}
\]

\[
\text{term } (\langle s_1 | s_2 \rangle) — \text{State with locals from } s_1 \text{ and globals from } s_2
\]

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

\text{thm } \text{big-step.Scope}
5.3.1 Parameter Passing

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

\[
\text{context fixes } f :: \text{com begin term} (\text{imp} \ (r1, r2) = f(x1, x2, x3)) \text{end}
\]

5.4 Recursive procedures

\[
\text{term (PCall "name") thm big-step.PCall}
\]

5.4.1 Procedure Scope

Execute command with local set of procedures

\[
\text{term (PScope } \pi \ c \text{) thm big-step.PScope}
\]

5.4.2 Syntactic sugar for procedure call with parameters

\[
\text{term (imp} (r1, r2) = \text{rec name}(x1, x2, x3))
\]

5.5 More Readable VCs

\[
\text{lemmas nat-distribs = nat-add-distrib nat-diff-distrib Suc-diff-le nat-mult-distrib nat-div-distrib}
\]

\[
\text{lemma } s_0 \ "n" \ 0 \geq 0 \implies \text{ wlp } \pi \ \text{exp-count-up1} \ (\lambda s \ . \ "a" \ 0 = 2^\text{nat} (s_0 \ "a" \ 0)) \ s_0
\]

\[
\text{unfolding exp-count-up1-def}
\]

\[
\text{apply } (\text{subst annotate-whileI[where)
\]

\[
I = \lambda s, s \ "n" \ 0 = s_0 \ "n" \ 0 \land s \ "a" \ 0 = 2^\text{nat} (s \ "c" \ 0) \land 0 \leq s \ "c" \ 0 \land s \ "c" \ 0 \leq s_0 \ "n" \ 0
\]

\[
]\]

\[
\text{apply (i-vcg-preprocess; i-vcg-gen; clarsimp)}
\]

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

\[
\text{apply i-vcg-postprocess}
\]

\[
\text{by (auto simp: algebra-simps nat-distribs)}
\]

\[
\text{lemma } s_0 \ "n" \ 0 \geq 0 \implies \text{ wlp } \pi \ \text{exp-count-up1} \ (\lambda s \ . \ "a" \ 0 = 2^\text{nat} (s_0 \ "a" \ 0)) \ s_0
\]

\[
\text{unfolding exp-count-up1-def}
\]

\[
\text{apply } (\text{subst annotate-whileI[where)
\]

\[
I = \lambda s, s \ "n" \ 0 = s_0 \ "n" \ 0 \land s \ "a" \ 0 = 2^\text{nat} (s \ "c" \ 0) \land 0 \leq s \ "c" \ 0 \land s \ "c" \ 0 \leq s_0 \ "n" \ 0
\]
The postprocessor is invoked by default

```text
apply vcg
oops
```

## 5.6 Specification Commands

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

Old way of proving a specification:

```text
lemma let n = s "n" 0 in n ≥ 0
⇒ wlp π exp-count-upI (λs. let a = s "a" 0; n₀ = s "n" 0 in
a = 2 ^ nat (n₀)) s₀
unfolding exp-count-up1-def
apply (subst annotate-whileI [where
I = λs. s "n" 0 = s₀ "n" 0 ∧ s "a" 0 = 2 ^ nat (s "c" 0)
∧ 0 ≤ s "c" 0 ∧ s "c" 0 ≤ s₀ "n" 0
])
apply vcg
apply (auto simp: algebra-simps nat-distrib)
done

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

IMP2 specification commands

```text
program-spec (partial) exp-count-up
assumes 0 ≤ n — Precondition. Use variable names of program.
ensures a = 2 ^ nat n₀ — Postcondition. Use variable names of programs. Suffix with _₀ to refer to initial state
defines — Program
{
  a = 1;
  c = 0;
  while (c < n)
    @invariant (n=n₀ ∧ a=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
    }
}
apply vcg
```
by (auto simp: algebra-simps nat-distribs)

**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^n nat n_0
**defines**
\[
\begin{align*}
  & a = 1; \\
  & c = 0; \\
  & \text{while (c} \lt n) \\
  & \qquad \text{invariant } (n\equiv n_0 \land a=2^n \land 0\leq c \land c\leq n) \\
  & \qquad \text{variant } (n-c) \\
  & \{ \\
  & \quad a=2*a; \\
  & \quad c=c+1 \\
  & \}
\end{align*}
\]
**apply** vcg
by (auto simp: algebra-simps nat-distribs)

**Simple Recursion**

**recursive-spec** exp-rec(n) returns a **assumes** 0\leq n **ensures** a=2^n nat n_0 **variant** n
**defines** (if (n=0) a=1 else \{t=rec exp-rec(n-1); a=2*t\})
**apply** vcg
**apply** (auto simp: algebra-simps nat-distribs)
by (metis Suc-le-D diff-Suc-Suc dvd-1-left dvd-imp-le minus-nat.diff-0 nat-0-iff nat-int.neq0_conv of-nat-0 order-class.order.antisym pos-int-cases power-Suc zless-nat-eq-int-zless)

**Mutual Recursion:** See Examples

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

**6 Examples**

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

43
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

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

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

```plaintext
program-spec sq-prog
  assumes $n \geq 0$ ensures $a = n_0 \cdot n_0$
  defines (;
    $a = 0$;
    $z = 1$;
```
\(i = 0\);
while \((i < n)\)
  @variant \((n_0 - i)\)
    @invariant \((n_0 = n \land 0 \leq i \land i \leq n \land a = i \cdot i \land z = 2 \cdot i + 1)\)
  \{
    a = a + z;
    z = z + 2;
    i = i + 1
  \}
by vcg-cs (simp add: algebra-simps)

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

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 \((n_0 = n \land 1 \leq i \land i \leq n + 1 \land a = factorial (i - 1))\)
  \{
    a = a \cdot i;
    i = i + 1
  \}
by vcg (simp add: antisym-conv)

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

lemma fib-simps[simp]:
  \(i \leq 0 \implies fib i = 0\)
  \(i = 1 \implies fib i = 1\)
  \(i > 1 \implies fib i = fib (i - 2) + fib (i - 1)\)
by simp+

lemmas [simp del] = fib.simps

With precondition

program-spec fib-prog
  assumes \(n \geq 0\) ensures \(a = fib n\)
defines \(\)
\[a = 0; b = 1;\]
\[i = 0;\]
\[\text{while } (i < n)\]
\[\text{variant } (n_0 - i)\]
\[\text{invariant } (n = n_0 \land 0 \leq i \land i \leq n \land a = \text{fib } i \land b = \text{fib } (i + 1)):\]
\[\{\]
\[c = b;\]
\[b = a + b;\]
\[a = c;\]
\[i = i + 1\]
\[\}\]
\text{by vcg-cs (simp add: algebra-simps)}

Without precondition, returning 0 for negative numbers

\textbf{program-spec} \textit{fib-prog' \`}
\textbf{assumes} True \textbf{ensures} \textit{a} = \textit{fib } n_0
\textbf{defines} (i
\[a = 0; b = 1;\]
\[i = 0;\]
\[\text{while } (i < n)\]
\[\text{variant } (n_0 - i)\]
\[\text{invariant } (n = n_0 \land (0 \leq i \land i \leq n \land n_0 < 0 \land i = 0) \land a = \text{fib } i \land b = \text{fib } (i + 1)):\]
\[\{\]
\[c = b;\]
\[b = a + b;\]
\[a = c;\]
\[i = i + 1\]
\[\}\]
\text{by vcg-cs (auto simp: algebra-simps)}

\textbf{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.

\textbf{program-spec} \textit{exp-count-down}
\textbf{assumes} \( 0 \leq n \)
\textbf{ensures} \textit{a} = \textit{2 } ^ \textit{nat } n_0
\textbf{defines} (i
\[a = 1;\]
\[\text{while } (n > 0)\]
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.

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 $\forall h \ l \ n_0 : \text{int}
\begin{align*}
&\forall \text{"invariants"; } 0 \leq n_0; \sim 1 + l < h; 0 \leq l; l < h; l \times l \leq n_0; \\
&n_0 < h \times h
\end{align*}
\implies n_0 < 1 + (l \times l + l \times 2)
by (smt mult.commute semiring-normalization-rules(3))

program-spec sqr-bisect
assumes $0\leq n$ ensures $r^2 \leq n_0 \land n_0 < (r+1)^2$
defines
\begin{align*}
l &\leftarrow 0; h \leftarrow n + 1; \\
\text{while } (l + 1 < h)
&\begin{align*}
&\@\text{variant } (h - l)
&\@\text{invariant } (0 \leq l \land l \times h \land l^2 \leq n \land n < h^2)
&\begin{align*}
&m = (l + h) / 2;
&\text{if } (m \times m \leq n) l = m \text{ else } h = m
\end{align*}
&
\end{align*}
&
\); \\
&r \leftarrow l
\)
apply vcg

We use quick-and-dirty apply style proof to discharge the VCs
\begin{align*}
&\text{apply (auto simp: power2-eq-square algebra-simps add-pos-pos)} \\
&\text{apply (smt not-sum-squares-lt-zero)} \\
&\text{by (smt mult.commute semiring-normalization-rules(3))}
\end{align*}

6.2 Debugging

6.2.1 Testing Programs

Stepwise

schematic-goal Map.empty: (sqr-approx-below, $\"n\":\lambda -.4$) $\Rightarrow$ ?s
unfolding sqr-approx-below-def
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
done

Or all steps at once

schematic-goal Map.empty: (sqr-bisect, $\"n\":\lambda -.4900000001$) $\Rightarrow$ ?s
unfolding sqr-bisect-def
by big-step
6.3 More Numeric Algorithms

6.3.1 Euclid’s Algorithm (with subtraction)

thm gcd.commute gcd-diff1

program-spec euclid1
  assumes $a > 0 \land b > 0$
  ensures $a = \gcd a \ 0 \ b \ 0$
  defines ( 
    while ($a \not= b$) 
      @invariant ($\gcd a \ b = \gcd a \ 0 \ b \ 0 \ \land (a > 0 \ \land b > 0)$) 
      @variant ($a + b$) 
      { 
        if ($a < b$) $b = b - a$
        else $a = a - b$
      } 
    )
  apply vcg-cs
  apply (metis gcd.commute gcd-diff1)
  apply (metis gcd-diff1)
  done

6.3.2 Euclid’s Algorithm (with mod)

thm gcd-red-int[symmetric]

program-spec euclid2
  assumes $a > 0 \land b > 0$
  ensures $a = \gcd a \ 0 \ b \ 0$
  defines ( 
    while ($b \not= 0$) 
      @invariant ($\gcd a \ b = \gcd a \ 0 \ b \ 0 \ \land b \geq 0 \ \land a > 0$) 
      @variant ($b$) 
      { 
        $t = a$;
        $a = b$;
        $b = t \mod b$
      } 
    )
  apply vcg-cs
  by (simp add: gcd-red-int[symmetric])

6.3.3 Extended Euclid’s Algorithm

locale extended-euclid-aux-lemmas begin

lemma aux2:
  fixes $a \ b :: \text{int}$
  assumes $b = t \ast b_0 + s \ast a_0 \ q = a \div b \ \gcd a \ b = \gcd a_0 \ b_0$

49
shows $\gcd b \ (a - (a_0 \ast (s \ast q) + b_0 \ast (t \ast q))) = \gcd a_0 \ b_0$

proof -
have $a - (a_0 \ast (s \ast q) + b_0 \ast (t \ast q)) = a - b \ast q$
  unfolding $\langle b = \_ \rangle$ by (simp add: algebra-simps)
also have $a - b \ast q = a \mod b$
  unfolding $\langle q = \_ \rangle$ by (simp add: algebra-simps)
finally show ?thesis
  unfolding $\langle \gcd a \ b = \_ \rangle$ by (simp add: gcd-red-int[symmetric])
qed

lemma aux3:
  fixes $a \ b :: \mathit{int}$
  assumes $b = t \ast b_0 + s \ast a_0 \ q = a \div b \ b > 0$
  shows $t \ast (b_0 \ast q) + s \ast (a_0 \ast q) \leq a$
proof -
  have $t \ast (b_0 \ast q) + s \ast (a_0 \ast q) = q \ast b$
    unfolding $\langle b = \_ \rangle$ by (simp add: algebra-simps)
then show ?thesis
  using $\langle b > 0 \rangle$
    by (simp add: algebra-simps $\langle q = \_ \rangle$
      (smt Euclidean-Division.pos-mod-sign cancel-div-mod-rules(1)
      mult.commute))
qed

end

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

program-spec euclid-extended
assumes $a > 0 \land b > 0$
ensures $\old-r = \gcd a_0 \ b_0 \land \gcd a_0 \ b_0 = a_0 \ast \old-s + b_0 \ast \old-t$
defines $\old-r$
$s = 0; \ \old-s = 1;$
$t = 1; \ \old-t = 0;$
$r = b; \ \old-r = a;$
while ($r \neq 0$)
  @invariant $
  \gcd \old-r \ r = \gcd a_0 \ b_0 \land r \geq 0 \land \old-r > 0$
  $\land a_0 \ast \old-s + b_0 \ast \old-t = \old-r \land a_0 \ast s + b_0 \ast t = r$
  $
  \langle \old-r \rangle$
  $
  \$
  @variant ($r$)
  $
  $quotient = \old-r \ / \ r;$
  $temp = \old-r;$
  $\old-r = r;$
  $r = temp - quotient \ast r;$
  $temp = \old-s;$
  $
proof –
interpret extended-euclid-aux-lemmas .
show ?thesis
  apply vcg-cs
  apply (simp add: algebra-simps)
  apply (simp add: aux2 aux3 minus-div-mult-eq-mod)+
done
qed

Non-Wikipedia version

context extended-euclid-aux-lemmas begin
lemma aux:
  fixes a b q x y :: int
  assumes a = old-y * b + old-x * a
  b = y * b + x * a
  q = a div b
  shows a mod b + (a0 * (x * q) + b0 * (y * q)) = a
proof –
  have *: a0 * (x * q) + b0 * (y * q) = q * b
    unfolding ⟨q = -⟩ by simp
  show ?thesis
    unfolding * unfolding ⟨q = -⟩ by simp
qed

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;
\begin{verbatim}
t = a;
a = b;
b = t \mod b;
t = x;
x = old-x - q \times x;
old-x = t;
t = y;
y = old-y - q \times y;
old-y = t
\}
x = old-x;
y = old-y
\end{verbatim}

\textbf{proof} –
\begin{itemize}
\item interpret \texttt{extended-euclid-aux-lemmas}.
\item show \texttt{?thesis}
\item apply \texttt{vcg-cs}
\item apply (\texttt{simp add: gcd-red-int[symmetric]})
\item apply (\texttt{simp add: algebra-simps})
\item apply (rule aux; simp add: algebra-simps)
\item done
\end{itemize}
qed

\textbf{6.3.4 Exponentiation by Squaring}

\textbf{lemma \texttt{ex-by-sq-aux}}:
\begin{itemize}
\item fixes \texttt{x :: int and n :: nat}
\item assumes \texttt{n \mod 2 = 1}
\item shows \texttt{x \ast (x \ast x) \hat{\ast} (n \Div 2) = x \hat{\ast} n}
\end{itemize}
\textbf{proof} –
\begin{itemize}
\item have \texttt{n \> 0}
\item using \texttt{assms by presburger}
\item have \texttt{2 \ast (n \Div 2) = n - 1}
\item using \texttt{assms by presburger}
\item then have \texttt{(x \ast x) \hat{\ast} (n \Div 2) = x \hat{\ast} (n - 1)}
\item by (\texttt{simp add: semiring-normalization-rules})
\item with \texttt{0 < n} show \texttt{?thesis}
\item by \texttt{simp (metis Suc-pred power.simps(2))}
\end{itemize}
qed

A classic algorithm for computing \(x^n\) works by repeated squaring, using the following recurrence:
\begin{itemize}
\item \(x^n = x \ast x^{(n-1)/2}\) if \(n\) is odd
\item \(x^n = x^{n/2}\) if \(n\) is even
\end{itemize}

\textbf{program-spec \texttt{ex-by-sq}}
\begin{itemize}
\item assumes \texttt{n \geq 0}
\item ensures \texttt{r = x_0 \hat{\ast} nat n_0}
\end{itemize}
defines:
\[ r = 1; \]
while \( n \neq 0 \)
    @invariant:
        \( n \geq 0 \land r \cdot x \cdot \text{nat } n = x_0 \cdot \text{nat } n_0 \)
    @variant \( n \)
    \{
        if \( (n \mod 2 == 1) \) \{
            r = r \cdot x
        \}
        x = x \cdot x;
        n = n \div 2
    \}
apply \ vcg-cs
subgoal
apply ( subst (asm) ex-by-sq-aux [symmetric])
apply (smt Suc-1 nat-1 nat-2 nat-mod-distrib nat-one-as-int)
by ( simp add: div-int-pos-iff int-div-less-self nat-div-distrib)
apply ( simp add: algebra-simps semiring-normalization-rules nat-mult-distrib; fail)
apply ( simp add: algebra-simps semiring-normalization-rules nat-mult-distrib; fail)
done

6.3.5 Power-Tower of 2s

fun tower2 where
    tower2 0 = 1
| tower2 (Suc n) = 2 \cdot tower2 n
definition tower2' n = int (tower2 (nat n))

program-spec tower2-imp
assumes \( \langle m > 0 \rangle \)
ensures \( \langle a = tower2' m_0 \rangle \)
defines:
    \( a = 1; \)
    while \( m > 0 \)
        @variant \( m \)
        @invariant \( \langle 0 \leq m \land m \leq m_0 \land a = tower2' (m_0 - m) \rangle \)
        \{
            n = a;
            a = 1;
            while \( n > 0 \)
                @variant \( n \)
                @invariant \( \langle \text{True} \rangle \) — This will get ugly, there is no \( n_0 \) that we

53
could use!
\[
\begin{align*}
  \{ & \quad a=2*a; \\
  & \quad n=n-1 \\
  \} \\
\end{align*}
\]

\[ m=m-1 \]

\textbf{oops}

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

\begin{verbatim}
program-spec tower2-imp
  assumes \( m>0 \)
  ensures \( a = \text{tower2}' \ m_0 \)
  defines \( a=1; \)
    while \( m>0 \)
      @variant \( \langle m \rangle \)
      @invariant \( 0 \leq m \land m \leq m_0 \land a = \text{tower2}' (m_0-m) \)
      \{
        n=a;  \\
        inline exp-count-down;  \\
        m=m-1
      \}
  \}
\end{verbatim}

apply \textit{vcg-cs}
by \texttt{(auto simp: algebra-simps tower2'-def nat-distrib)}

6.4 Array Algorithms

6.4.1 Summation

\begin{verbatim}
program-spec array-sum
  assumes \( l \leq h \)
  ensures \( r = (\sum_{i=l_0..<h_0.} a_0 \ i) \)
  defines \( r = 0; \)
    while \( l<h \)
      @invariant \( l_0 \leq l \land l \leq h \land r = (\sum_{i=l_0..<l.} a_0 \ i) \)
      @variant \( h-l \)
      \{
        r = r + a[l]; \\
        l=l+1
      \}
  \}
\end{verbatim}

apply \textit{vcg-cs}
by \texttt{(auto simp: intvs-incdec)}
6.4.2 Finding Least Index of Element

program-spec find-least-idx
assumes \( l \leq h \)
ensures \( \text{if } l = h_0 \text{ then } x_0 \notin a_{0'} \{ l_0..< h_0 \} \text{ else } l \in \{ l_0..< h_0 \} \land a_0 \ l = x_0 \land \ x_0 \notin a_{0'} \{ l_0..< l \} \) 
defines (while \( l < h \land a[l] \neq x \))
@variant \( h - l \)
@invariant \( l_0 \leq l \land l \leq h \land x \notin a' \{ l_0..< l \} \);
\( l = l + 1 \)
)
apply vcg-cs
by (smt atLeastLessThan-iff imageI)

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 \( l < h \land a[l - 1] \leq a[l] \)
@variant \( h - l \)
@invariant \( l_0 < l \land l \leq h \land \text{ran-sorted } a_0 \ l \);
\( l = l + 1 \);

if \( l = h \) \( r = 1 \) else \( r = 0 \)
)
apply vcg-cs
apply (auto simp: ran-sorted-def)
by (smt atLeastLessThan-iff)

6.4.4 Find Equilibrium Index

definition is-equil \( a \ l \ h \ i \equiv l \leq i \land i < h \land ( \sum j = l..<i. \ a \ j ) = ( \sum j = i..<h. \ a \ j ) \)

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

apply vcg-cs
apply (auto simp: intvs-incdec is-equil-def)
apply (metis atLeastLessThan-iff eq-iff finite-atLeastLessThan-int
sum-diff1)
apply force
by force

6.4.5 Rotate Right

program-spec rotate-right
  assumes 0 < n
  ensures \(\forall i \in \{0..<n\}. \ a \ i = a_0 ((i - 1) \mod n)\)
  defines
  \(i = 0;\)
  \(\text{prev} = a[n - 1];\)
  while (i < n)
    @invariant
    \(\ 0 \leq i \land i \leq n\)
    \(\land (\forall j \in \{0..<i\}. \ a \ j = a_0 ((j - 1) \mod n))\)
    \(\land (\forall j \in \{i..<n\}. \ a \ j = a_0 \ j)\)
    \(\land \text{prev} = a_0 ((i - 1) \mod n)\)
  }\
  @variant \(n - i\)
  {
    \(\text{temp} = a[i];\)
    \(a[i] = \text{prev};\)
    \(\text{prev} = \text{temp};\)
    \(i = i + 1\)
  }
apply vcg-cs
6.4.6 Binary Search, Leftmost Element

We first specify the pre- and postcondition

**definition** bin-search-pre a l h ←→ l ≤ h \(\land\) ran-sorted a l h

**definition** bin-search-post a l h x i ←→

\(l \leq i \land i \leq h \land (\forall i \in \{l..<i\}. a i < x) \land (\forall i \in \{i..<h\}. x \leq a i)\)

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 \(a_0\) l \(h_0\) \(x_0\) l}\rangle\)

defines :

\[
\text{while } (l < h) \\
\text{variant } (h - l) \\
\text{invariant } (l_0 \leq l \land l \leq h \land h \leq h_0 \land (\forall i \in \{l_0..<l}\}. a i < x) \land (\forall i \in \{h..<h_0\}. x \leq a i) \\
\{ \\
m = \frac{(l + h)}{2}; \\
\text{if } (a[m] < x) l = m + 1 \\
\text{else } h = m \\
\}
\]

**apply** vcg-e

**apply** (auto simp: algebra-simps bin-search-pre-def bin-search-post-def)

Driving sledgehammer to its limits ...

**apply** (smt div-add-self1 even-succ-div-two odd-two-times-div-two-succ ran-sorted-alt)

_by_ (smt div-add-self1 even-succ-div-two odd-two-times-div-two-succ ran-sorted-alt)

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: \(x \in a'\{l..<h\} \leftrightarrow (i < h \land x = a i)\)

_and_ bin-search-leftmost: \(x \notin a'\{l..<i\}\)

_using assms apply _–_

**apply** (auto simp: bin-search-post-def bin-search-pre-def)

**apply** (smt atLeastLessThan-iff ran-sorted-alt)

**done**

6.4.7 Naive String Search

**program-spec** match-string
assumes \( l_1 \leq h_1 \)

ensures \( \forall j \in \{0..<i\}. \ a(l + j) = b(l + j) \land (i < h_1 - l_1 \longrightarrow a(l + i) \neq b(l + i)) \)

\( \land 0 \leq i \land i \leq h_1 - l_1 \)

defines \( i \)

\( i = 0; \)

while \( (l_1 + i < h_1 \land a[l + i] = b[l + i]) \)

\( \land \forall j \in \{0..<i\}. \ a(l + j) = b(l + j) \land 0 \leq i \land i \leq h_1 - l_1 \)

\( \implies a(l + i) \neq b(l + i) \)

\( \land 0 \leq i \land i \leq h_1 - l_1 \)

by vcg-cs auto

lemma \( \text{lran-eq-iff}' \): \( \text{lran} \ a \ l_1 \ (l_1 + (h - l)) = \text{lran} \ a' \ l \ h \)

\( \iff (\forall i. \ 0 \leq i \land i < h - l \longrightarrow a(l + i) = a'(l + i)) \) if \( l \leq h \)

using that

proof (induction nat \((h - l)\) arbitrary: \( h \))

case \( 0 \)

then show \( ?\)case

by auto

next

case \( \text{Suc } x \)

then have \( \ast: \ x = \text{nat} \ (h - 1 - l) \ l \leq h - 1 \)

by auto

note \( \ast\ \text{IH} = \text{Suc} \ \text{hyps}(1)[OF \ \ast] \)

from \( \ast\ \text{have} 1: \)

\( \text{lran} \ a \ l_1 \ (l_1 + (h - l)) = \text{lran} \ a \ l_1 \ (l_1 + (h - 1 - l)) \) @ \( [a(l + h - 1 - l)] \)

\( \text{lran} \ a' \ l \ h = \text{lran} \ a' \ l \ (h - 1) \) @ \( [a'(h - 1)] \)

by \( \text{(auto simp: lran-bwd-simp algebra-simps lran-butlast[symmetric])} \)

from \( \ast\ \text{IH} = \text{Suc} \ \text{hyps}(2) \text{ Suc.prems show} \ ?\)case

unfolding \( 1 \)

apply auto

subgoal for \( i \)

by \( \text{(cases } i = h - 1 - l) \text{ auto} \)

done

qed

program-spec \( \text{match-string}' \)

assumes \( l_1 \leq h_1 \)

ensures \( i = h_1 - l_1 \iff \text{lran} \ a \ l \ (l_1 + (h_1 - l_1)) = \text{lran} \ b \ l_1 \ h_1 \)

for \( i \ h_1 \ l_1 \ l \ a [] b [] \)

defines \( \text{inline match-string} \)

by vcg-cs \( \text{(auto simp: lran-eq-iff') } \)

program-spec \( \text{substring} \)
assumes $l \leq h \land ll \leq h1$
ensures $\text{match} = 1 \iff (\exists \ j \in \{l_0..<h0\}. \ \text{ran} \ a \ j \ (j + (h1 - ll)) = \text{ran} \ b \ ll \ h1)$

for $a[] \ b[]$
defines $i$
  match $= 0$;
  while ($l < h \land \text{match} == 0$)
    @invariant: $l_0 \leq l \land l \leq h \land \text{match} \in \{0,1\} \land$
      (if match $= 1$
        then ran a l $(l + (h1 - ll)) = \text{ran} \ b \ ll \ h1 \land l < h$
        else $(\forall \ j \in \{l_0..<l\}. \ \text{ran} \ a \ j \ (j + (h1 - ll)) \neq \text{ran} \ b \ ll \ h1)$)
    @variant: $(h - l) * (1 - \text{match})$
    { inline match-string';
      if ($i == h1 - ll$) \{ match $= 1$ \}
      else \{ $l = l + 1$ \}
    }
  }
by vcg-cs auto

program-spec substring'
assumes $l \leq h \land ll \leq h1$
ensures $\text{match} = 1 \iff (\exists \ j \in \{l_0..h0-(h1 - ll)\}. \ \text{ran} \ a \ j \ (j + (h1 - ll)) = \text{ran} \ b \ ll \ h1)$

for $a[] \ b[]$
defines $i$
  match $= 0$;
  if ($l + (h1 - ll) \leq h$) \{ 
    $h = h - (h1 - ll) + 1$;
    inline substring 
  \}
by vcg-cs auto

program-spec substring''
assumes $l \leq h \land ll \leq h1$
ensures $\text{match} = 1 \iff (\exists \ j \in \{l_0..<h0-(h1 - ll)\}. \ \text{ran} \ a \ j \ (j + (h1 - ll)) = \text{ran} \ b \ ll \ h1)$

for $a[] \ b[]$
defines $i$
  match $= 0$;
  if ($l + (h1 - ll) \leq h$) \{ 
    while ($l + (h1 - ll) < h \land \text{match} == 0$)
      @invariant: $l_0 \leq l \land l \leq h - (h1 - ll) \land \text{match} \in \{0,1\} \land$
        (if match $= 1$
          then ran a l $(l + (h1 - ll)) = \text{ran} \ b \ ll \ h1 \land l < h - (h1 - ll)$
          else $(\forall \ j \in \{l_0..<l\}. \ \text{ran} \ a \ j \ (j + (h1 - ll)) \neq \text{ran} \ b \ ll \ h1)$)
  \}
59
@variant \((h - l) \ast (1 - \text{match})\)
{
  \text{inline \text{match-string}'};
  \text{if} \ (i == h1 - l1) \ \{\text{match} = 1\}
  \text{else} \ \{l = l + 1\}
}

by \text{vec-cs auto}

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

\text{lemma \text{tran-eq-append-iff}:}
\text{tran} \ a \ l \ h = \text{as} @ \text{bs} \iff (\exists \ i, l \leq i \land i \leq h \land \text{as} = \text{tran} \ a \ i \ l
\land \text{bs} = \text{tran} \ a \ i \ h) \ \text{if} \ l \leq h
\text{apply safe}

\text{subgoal}
\text{using that}
\text{proof (induction as arbitrary: l)}
\text{case Nil}
\text{then show ?case}
\text{by auto}
\text{next}
\text{case (Cons \(x\) \(\text{as}\))}
\text{from this(2 \text{--}) have tran} \ a \ (l + 1) \ h = \text{as} \ @ \text{bs} \ a \ l = x \ l + 1 \leq h
\text{apply -}
\text{subgoal}
\text{by simp}
\text{subgoal}
\text{by (smt append-Cons list.inject tran.simps tran-append1)}
\text{subgoal}
\text{using add1-zle-eq by fastforce}
\text{done}
\text{from Cons.IH[OF this(1,3)] guess i by safe}
\text{note IH = this}
\text{with (a l = x) show ?case}
\text{apply (intro exI[where \(x = i\)])}
\text{apply auto}
\text{by (smt IH(3) tran-prepend1)}
\text{qed}
\text{apply (subst tran-split; simp)}
\text{done}

\text{lemma \text{tran-split'}:}
(\exists j \in\{l..h - (h1 - l1)\}. \text{tran} \ a \ j \ (j + (h1 - l1)) = \text{tran} \ b \ l1 \ h1)
= (\exists \text{as bs}, \text{tran} \ a \ l \ h = \text{as} \ @ \text{tran} \ b \ l1 \ h1 \ @ \text{bs}) \ \text{if} \ l \leq h \ l1 \leq h1

60
proof safe
fix j
assume j: j ∈ {l..h − (h1 − l1)} and match: lran a j (j + (h1 − l1)) = lran b l h1
with l1 ≤ h1 have lran a l h = lran a l j @ lran a j (j + (h1 − l1)) @ lran a (j + (h1 − l1)) h
apply (subst lran-split[where p = j], simp, simp)
apply (subst (2) lran-split[where p = j + (h1 − l1)]; simp)
done
then show ∃as bs. lran a l h = as @ lran b l1 h1 @ bs
by (auto simp: match)
next
fix as bs
assume lran a l h = as @ lran b l1 h1 @ bs
with that lran-eq-append-iff[of l h a as lran b l1 h1 @ bs]
obtain i
where
as = lran a l i iran a i h = lran b l1 h1 @ bs i ≤ i ≤ h by auto
with lran-eq-append-iff[of i h a lran b l1 h1 bs]
obtain j where j:
irn b l1 h1 = lran a i j bs = lran a j h i ≤ j j ≤ h by auto
moreover have j = i + (h1 − l1)
proof −
have length (lran b l1 h1) = nat (h1 − l1) length (lran a i j) = nat (j − i)
by auto
with j (1,3) that show ?thesis
by auto
qed
ultimately show ∃j∈{l..h − (h1 − l1)}. lran a j (j + (h1 − l1)) = lran b l1 h1
using (l ≤ i) by auto
qed

program-spec substring-final
assumes l ≤ h ∧ 0 ≤ len
ensures match = 1 ←→ (∃as bs. lran a l0 h0 = as @ lran b 0 len @ bs)
for l h len match a[] b[]
defines l1 = 0; h1 = len; inline substring';
supply [simp] = lran-split[symmetric]
apply vcg-cs
done

6.4.8 Insertion Sort

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

**definition** \( \text{insort-insert-post } l \ j \ a_0 \ a \ i \)

\( \leftarrow \{ \text{let key} = a_0 \ j \text{ in} \right. \)

\( i \in \{ l-1..<j \} \quad \text{— } i \text{ is in range} \)

\( \quad \text{— Content of new array} \)

\( \land (\forall k \in \{l..i\}. \ a k = a_0 \ k) \)

\( \land a \ (i+1) = \text{key} \)

\( \land (\forall k \in \{i+2..j\}. \ a k = a_0 \ (k-1)) \)

\( \land a = a_0 \text{ on } \{l..j\} \)

\( \quad \text{— Placement of key} \)

\( \land (i \geq l \rightarrow a i \leq \text{key}) \quad \text{— Element at } i \text{ smaller than key, if it exists} \)

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

\( \)

for \( l \ j \ i :: \text{int } \) and \( a_0 \ a :: \text{int } \Rightarrow \text{int } \)

**program-spec** \( \text{insort-insert} \)

assumes \( l < j \)

ensures \( \text{insort-insert-post } l \ j \ a_0 \ a \ i \)

defines

\( \quad \text{key} = a[j]; \)

\( i = j-1; \)

\( \text{while } (i \geq l \land a[i] > \text{key}) \)

\( \text{@variant } (i-l+1) \)

\( \text{@invariant } l-1 \leq i \land l < j \)

\( \quad \land (\forall k \in \{l..i\}. \ a k = a_0 \ k) \)

\( \quad \land (\forall k \in \{i+2..j\}. \ a k > \text{key} \land a k = a_0 \ (k-1)) \)

\( \quad \land a = a_0 \text{ on } \{l..j\} \)

\( \}

\( \quad \{ \quad a[i+1] = a[i]; \)

\( \quad \quad i = i-1 \)

\( \quad \} ; \)

\( a[i+1] = \text{key} \}

unfolding \( \text{insort-insert-post-def Let-def} \)

apply \( \text{vcg} \)

apply \( \text{auto} \)

by \( \text{(smt atLeastAtMost-iff)} \)

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

**Invoking** \( \text{insort-insert} \) will sort in the element

**lemma** \( \text{insort-insert-sorted} \):

assumes \( l < j \)

assumes \( \text{insort-insert-post } l \ j \ a \ a' \ i' \)

assumes \( \text{ran-sorted } a \ l \ j \)

62
Invoking \textit{insort-insert} will only mutate the elements

\textbf{lemma} \textit{insort-insert-ran1}:

\begin{itemize}
  \item \texttt{assumes} \( \langle l<j \rangle \)
  \item \texttt{assumes} \( \textit{insort-insert-post} \ l j a \ a' i \)
  \item \texttt{shows} \( \textit{mset-ran} \ a' \{l..j\} = \textit{mset-ran} \ a \{l..j\} \)
\end{itemize}

\textbf{proof} –

\begin{itemize}
  \item \texttt{from} \(\texttt{assms}\) \texttt{have} \(\texttt{EQS}:\)
    \begin{itemize}
      \item \(a' = a \ \texttt{on} \ \{l..i\}\)
      \item \(a'(i+1) = a \ j\)
      \item \(a' = (a \ o (+)(-1)) \ \texttt{on} \ \{i+2..j\}\)
    \end{itemize}
  \item \texttt{from} \(\texttt{assms}\) \texttt{have} \(\texttt{ranshift} : \textit{mset-ran} \ (a \ o (+)(-1)) \ \{i+2..j\} = \textit{mset-ran} \ a \{i+1..j-1\}\)
    \begin{itemize}
      \item \texttt{by} \(\texttt{(simp add: mset-ran-shift algebra-simps)}\)
    \end{itemize}
  \item \texttt{have} \(\textit{mset-ran} \ a' \{l..j\} = \textit{mset-ran} \ a' \{l..i\} + \{\# \ a' \ (i+1) \ \#\} + \textit{mset-ran} \ a' \{i+2..j\}\)
    \begin{itemize}
      \item \texttt{using} \(\langle l<j \rangle \ \langle l<i+1 \rangle \ \langle i+1<j \rangle\)
      \item \texttt{apply} \(\texttt{(simp add: mset-ran-combine)}\)
      \item \texttt{by} \(\texttt{(auto intro: arg-cong[where f=mset-ran a'])}\)
    \end{itemize}
  \item \texttt{also have} \(\ldots = \textit{mset-ran} \ a \{l..i\} + \{\# \ a \ j \ \#\} + \textit{mset-ran} \ (a \ o (+)(-1)) \ \{i+2..j\}\)
    \begin{itemize}
      \item \texttt{using} \(\texttt{EQS(1,3)[THEN mset-ran-cong]} \ \texttt{EQS(2) by simp}\)
      \item \texttt{also have} \(\ldots = \textit{mset-ran} \ a \{l..i\} + \textit{mset-ran} \ a \{i+1..j-1\} + \{\# \ a \ j \ \#\}\)
      \begin{itemize}
        \item \texttt{by} \(\texttt{(simp add: mset-ran-shift algebra-simps)}\)
      \end{itemize}
      \item \texttt{also have} \(\ldots = \textit{mset-ran} \ a \{l..j\}\)
        \begin{itemize}
          \item \texttt{using} \(\langle l<j \rangle \ \langle l<i+1 \rangle \ \langle i+1<j \rangle\)
          \item \texttt{apply} \(\texttt{(simp add: mset-ran-combine)}\)
          \item \texttt{by} \(\texttt{(auto intro: arg-cong[where f=mset-ran a']}}\)
        \end{itemize}
    \end{itemize}
  \item \(\texttt{finally show} \ ?\texttt{thesis} .\)
\end{itemize}

\texttt{qed}

The property \([?l < ?j; \textit{insort-insert-post} \ ?l \ ?j \ ?a \ ?a' ?i] \implies \ldots\)
mset-ran \(a'\) \{\(l\ldots j\)\} = mset-ran \(a\) \{\(l\ldots j\)\} extends to the whole array to be sorted

**lemma** \textit{insort-insert-ran2}:

- **assumes** \(l < j < h\)
- **assumes** \textit{insort-insert-post} \(l j a a' i\)
- **shows** \(mset-ran a'\) \{\(l\ldots h\)\} = \(mset-ran a\) \{\(l\ldots h\)\} (is \textit{thesis1})
  - and \(a'=a\) on \(\{l\ldots h\}\) (is \textit{thesis2})

**proof**

- from \textit{insort-insert-ran1} assms have \(mset-ran a'\) \{\(l\ldots j\)\} = \(mset-ran a\) \{\(l\ldots j\)\} by blast
- also from \{\textit{insort-insert-post} \(l j a a' i\) \textit{have} \(a'=a\) on \(\{j\ldots h\}\)\}
  - unfolding \textit{insort-insert-post-def Let-def} by (auto simp: eq-on-def)
  - hence \(mset-ran a'\) \{\(j\ldots h\)\} = \(mset-ran a\) \{\(j\ldots h\)\} by (rule \textit{mset-ran-cong})
- finally (mset-ran-combine-eqs) show \(\textit{thesis1}\)
  - by (simp add: assms ivl-disj-int-two (4) ivl-disj-un-two (4) le-less)
- from assms show \(\textit{thesis2}\)
  - unfolding \textit{insort-insert-post-def Let-def eq-on-def}
  - by auto

qed

Finally, we specify and prove correct the outer loop

**program-spec** \textit{insort}:

- **assumes** \(l < h\)
- **ensures** \textit{ran-sorted a l h} \& \(mset-ran a\) \{\(l\ldots h\)\} = \(mset-ran a_0\) \{\(l\ldots h\)\}
  - \& \(a=a_0\) on \(\{l\ldots h\}\)
  - for \(a[\] defines ( \(j = l + 1;\)
    - while \((j < h)\)
      - \textit{variant} \((h - j)\)
      - \textit{invariant} ( \(l < j \& j \leq h\) \& \textit{ran-sorted a l j}\)
        - \textit{Array is sorted up to j}
        - \& \textit{mset-ran a \{l\ldots h\} = mset-ran a_0 \{l\ldots h\}} \& \textit{Elements in range only permuted}
        - \& \(a=a_0\) on \(\{l\ldots h\}\)
      )
    )
  )
  }
  }

apply \textit{vvcg-cs}
apply (intro conjI)
subgoal by (rule \textit{insort-insert-sorted})
subgoal using \textit{insort-insert-ran2} \{1\} by auto
subgoal apply (frule (2) \textit{insort-insert-ran2} \{2\}) by (auto simp:
6.4.9 Quicksort

procedure-spec partition-aux\(a,l,h,p\) returns \((a,i)\)
assumes \(l \leq h\)
ensures \(\text{mset-ran } a_0 \{l_0..<h_0\} = \text{mset-ran } a \{l_0..<h_0\}\)
\(\land (\forall j \in \{l_0..<i\}, a j < p_0)\)
\(\land (\forall j \in \{i..<h_0\}, a j \geq p_0)\)
\(\land l_0 \leq i \land i \leq h_0\)
\(\land a_0 = a \text{ on } \{-l_0..<h_0\}\)
defines:
\(i = l; j = l;\)
while \((j < h)\)
\(@\text{invariant:}\)
\(l \leq i \land i \leq j \land j \leq h\)
\(\land \text{mset-ran } a_0 \{l_0..<h_0\} = \text{mset-ran } a \{l_0..<h_0\}\)
\(\land (\forall k \in \{i..<j\}, a k < p)\)
\(\land (\forall k \in \{i..<h\}, a k = a k)\)
\(\land a_0 = a \text{ on } \{-l_0..<h_0\}\)
\(@\text{variant:}\)
\(\langle h - j\rangle)\)
\(\{\)
\(\text{if } (a[j] < p) \{\text{temp } = a[i]; a[i] = a[j]; a[j] = \text{temp}; i = i + 1\};\)
\(j = j + 1\)
\}\n
supply \(\text{ran-eq-iff[simp] ran-tail[simp del]}\)
apply \(\text{vcg-cs}\)
subgoal by \((\text{simp add: mset-ran-swaps[unfolded swap-def]})\)
subgoal by \(\text{auto}\)
done

procedure-spec partition\(a,l,h,p\) returns \((a,i)\)
assumes \(l < h\)
ensures \(\text{mset-ran } a_0 \{l_0..<h_0\} = \text{mset-ran } a \{l_0..<h_0\}\)
\(\land (\forall j \in \{l_0..<i\}, a j < a i)\)
\(\land (\forall j \in \{i..<h_0\}, a j \geq a i)\)
\(\land l_0 \leq i \land i < h_0 \land a_0 (h_0 - 1) = a i\)
\(\land a_0 = a \text{ on } \{-l_0..<h_0\}\)
defines:
\(p = a[h - 1];\)
\((a,i) = \text{partition-aux}\(a,l,h - 1,p\);\)
\(a[h - 1] = a[i];\)
lemma quicksort-sorted-aux:
  assumes BOUNDS: l ≤ i i < h
  assumes LESS: ∀ j∈{l..<i}. a1 j < a1 i
  assumes GEQ: ∀ j∈{i..<h}. a1 i ≤ a1 j
  assumes R1: mset-ran a1 {l..<i} = mset-ran a2 {l..<i}
  assumes E1: a1 = a2 on −{l..<i}
  assumes SL: ran-sorted a2 l i
  assumes R2: mset-ran a2 {i + 1..<h} = mset-ran a3 {i + 1..<h}
  assumes E2: a2 = a3 on −{i + 1..<h}
  assumes SH: ran-sorted a3 (i + 1) h
  shows ran-sorted a3 l h
proof −
  have [simp]: {l..<i} ⊆ −{i + 1..<h} by auto
  have [simp]: a1 i = a3 i using E1 E2 by (auto simp: eq-on-def)

  note X1 = mset-ran-xfer-pointwise[where P = λx. x < p] for p, OF R1, simplified
  note X2 = eq-on-xfer-pointwise[where P = λx. x < p] for p, OF E2, of {l..<i}, simplified
  from LESS have LESS': ∀ j∈{l..<i}. a3 j < a3 i
    by (simp add: X1 X2)

  from GEQ have GEQ1: ∀ j∈{i+1..<h}. a1 i ≤ a1 j by auto
  have [simp]: {i + 1..<h} ⊆ −{l..<i} by auto
  note X3 = eq-on-xfer-pointwise[where P = λx. x ≥ p] for p, OF E1, of {i+1..<h}, simplified
  note X4 = mset-ran-xfer-pointwise[where P = λx. x ≥ p] for p, OF R2, simplified
  from GEQ1 have GEQ': ∀ j∈{i+1..<h}. a3 i ≤ a3 j by (simp add: X3 X4)

  from SL eq-on-xfer-ran-sorted[OF E2, of l i] have SL': ran-sorted
show thesis using combine-sorted-pivot[OF BOUNDS SL' SH LESS' GEQ']. qed

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

recursive-spec quicksort(a,l,h) returns a
  assumes True
  ensures ran-sorted a \( l_0 \ h_0 \) \& mset-ran a a \( l_0..<h_0\) = mset-ran a \( l_0..<h_0\) \& a_0=a on \( l_0..<h_0\)
  variant h-l
  defines (i = l; j = l;
  while (j < h)
  apply (vcg-cs; (intro conjI)?)
  subgoal using quicksort-sorted-aux by metis
  subgoal using quicksort-mset-aux by metis
  subgoal by (smt ComplD ComplI atLeastLessThan-iff eq-on-def)
  subgoal by (auto simp: ran-sorted-def)
  done

6.5 Data Refinement

6.5.1 Filtering

program-spec array-filter-negative
  assumes \( t \leq h \)
  ensures \( \text{ran a} \ l_0 \ i = \text{filter (} \lambda x. \ x \geq 0 \text{)} \) (\( \text{ran a_0} \ l_0 \ h_0 \))
  defines (i = l; j = l;
  while (j < h)
6.5.2 Merge Two Sorted Lists

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

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

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

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

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

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

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

```
lemma merge-eq: xs≠[] ∧ ys≠[] → merge xs ys = (if ys=[] ∨ (xs≠[] ∧ hd xs < hd ys) then hd xs # merge (tl xs) ys 
  else hd ys # merge xs (tl ys)
) 
  by (cases xs; cases ys; simp)
```
We do a first proof that our merge implementation on the arrays and indexes behaves like the functional merge on the corresponding lists.

The annotations use the $\text{lran}$ 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** $\text{merge-imp'}$

- **assumes** $l_1 \leq h_1 \land l_2 \leq h_2$
- **ensures** let $ms = \text{lran } m \ 0 \ j$; $xs_0 = \text{lran } a_1 \ 0 \ l_1 \ 0 \ h_1 \ 0$; $ys_0 = \text{lran } a_2 \ 0 \ l_2 \ 0 \ h_2 \ 0$
  
  $j \geq 0 \ \& \ ms = \text{merge } xs_0 \ ys_0$
- **defines** $j = 0$; while ($l_1 \neq h_1 \lor l_2 \neq h_2$)
  
  @variant $h_1 + h_2 - l_1 - l_2$
  
  @invariant let
  
  $xs = \text{lran } a_1 \ l_1 \ h_1$; $ys = \text{lran } a_2 \ l_2 \ h_2$; $ms = \text{lran } m \ 0 \ j$
  
  $xs_0 = \text{lran } a_1 \ 0 \ l_1 \ 0 \ h_1 \ 0$; $ys_0 = \text{lran } a_2 \ 0 \ l_2 \ 0 \ h_2 \ 0$
  
  in
  
  $l_1 \leq l_1 \land l_2 \leq h_2 \land 0 \leq j \land$
  
  merge $xs_0 \ ys_0 = ms \& \text{merge } xs \ ys$
  
  $j = j + 1$

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

apply $\text{vcg-cs}$

- subgoal apply (subst $\text{merge-eq}$) by auto
- subgoal by linarith
- subgoal apply (subst $\text{merge-eq}$) by auto
- done

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

**procedure-spec** merge-imp \((a_1,l_1,h_1,a_2,l_2,h_2)\) **returns** \((m,j)\)

**assumes** \(l_1 \leq h_1 \land l_2 \leq h_2 \land \text{sorted}(lran\ a_1\ l_1\ h_1) \land \text{sorted}(lran\ a_2\ l_2\ h_2)\)

**ensures** let \(ms = lran\ m\ 0\ j\) in

\[
\begin{align*}
    & j \geq 0 \\
    & \text{sorted}\ ms \\
    & \text{mset}\ ms = \text{mset}\ (lran\ a_1\ 0\ h_1) \land \text{mset}\ (lran\ a_2\ l_2\ h_2)
\end{align*}
\]

**for** \(l_1\ h_1\ l_2\ h_2\ a_1[\[]\ a_2[\[]\ m[\[]\ j\)

**defines** (`inline merge-imp'`)

**apply** vcg-cs

**apply** (auto simp: Let-def merge-mset dest: merge-sorted)

done

**thm** merge-imp-spec

**thm** merge-imp-def

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

\[
\begin{align*}
    & UNIV \cup a = UNIV \\
    & a \cup UNIV = UNIV
\end{align*}
\]

by auto

**lemma** merge-msets-aux: \([l \leq m; m \leq h, \implies mset\ (lran\ a\ l\ m) + mset\ (lran\ a\ m\ h) = mset\ (lran\ a\ l\ h)]\)

by (auto simp: mset-lran mset-ran-combine ivl-disj-un-two)

**recursive-spec** mergesort \((a,l,h)\) **returns** \((b,j)\)

**assumes** \(l \leq h\)

**ensures** \(0 \leq j \land \text{sorted}(lran\ b\ 0\ j) \land \text{mset}\ (lran\ b\ 0\ j) = \text{mset}(lran\ a_0\ l_0\ h_0)\)

**variant** \((h-l)\)

**for** \(a[\[]\ b[\[]\ defines\ (j\in\{0,1\})\)

\[
\begin{align*}
    & i f\ (l\in\{0\})\ j=0 \\
    & e l s e\ i f\ (l+1\in\{0\})\{ \\
    & \quad b[0] = a[l]; \\
    & \quad j=1 \\
    & e l s e\{ \\
    & \quad m = (h+l) / 2; \\
    & \quad (a_1,h_1) = \text{rec}\ \text{mergesort}\ (a,l,m); \\
    & \quad (a_2,h_2) = \text{rec}\ \text{mergesort}\ (a,m,h); \\
    & \quad (b,j) = \text{merge-imp}\ (a_1,0,h_1,a_2,0,h_2) \\
    & \}\}
\]

**apply** vcg
apply auto []
apply (auto simp: lran.simps) []
apply auto []
apply auto []
apply auto []
apply (auto simp: Let-def merge-msets-aux) []
done
print-theorems

6.5.3 Remove Duplicates from Array, using Bitvector Set

We use an array to represent a set of integers. If we only insert elements in range \{0..< n\}, this representation is called bit-vector (storing a single bit per index is enough).

definition set-of :: (int ⇒ 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) = {}
by auto

lemma set-of-insert [simp]: x ≠ 0 ⇒ set-of (a(i:=x)) = insert i (set-of a) by auto

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

lemma set-of-mem [simp]: i∈ set-of a ←→ a i ≠ 0 by auto
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=1; 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
}
apply vcg-cs
apply (auto simp: tran-eq-iff tran-upd-inside intro: arg-cong) [where f="R"] []
apply (auto simp: tran-eq-iff) []
done

procedure-spec bv-init () returns b
assumes True ensures (set-of b = {})
defines (clear b[])
by vcg-cs

procedure-spec bv-insert (x, b) returns b
assumes True ensures (set-of b = insert x_0 (set-of b_0))
defines (b[x] = 1)
by vcg-cs

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

procedure-spec bv-elem (x, b) returns r
assumes True ensures (r ≠ 0 \iff x_0 ∈ set-of b_0)
defines (r = b[x])
by vcg-cs

procedure-spec dedup ′ (a,l,h) returns (a,l,i)
assumes (l≤h) ensures (set (lran a l i) = set (lran a_0 l_0 h_0) ∧
distinct (lran a l i) ∧
set-of b = set (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 a_0 l_0 h_0)
∧ distinct (lran a l i)
∧ lran a j h = lran a_0 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)
\begin{verbatim}

apply vcg-cs
apply (auto simp: tran-eq-iff tran-upd-inside intro: arg-cong[where f=t])
done

6.6 Recursion

6.6.1 Recursive Fibonacci

recursive-spec fib-imp (i) returns r assumes True ensures \langle r = fib i0 \rangle
variant \langle i \rangle
defines :
  if (i\leq0) r=0
  else if (i=1) r=1
  else {
    r1 = rec fib-imp (i-2);
    r2 = rec fib-imp (i-1);
    r = r1+r2
  };

by vcg-cs

6.6.2 Homeier’s Cycling Termination

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

recursive-spec
  pedal (n,m) returns () assumes \langle n\geq0 \land m\geq0 \rangle
variant \langle n+m \rangle
defines :
  if (n\neq0 \land m\neq0) {
    G' = G + m;
    if (n<m) rec coast (n-1,m-1) else rec pedal(n-1,m)
  }
}

and

coast (n,m) returns () assumes \langle n\geq0 \land m\geq0 \rangle
variant \langle n+m+1 \rangle
defines :
  G = G + n;
  if (n<m) rec coast (n,m-1) else rec pedal (n,m)
}

by vcg-cs

\end{verbatim}
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))
subgoal by (cases m) auto
subgoal by (cases (m,n) rule: ack.cases) (auto)
done

recursive-spec relation less-than <(*lex*)> less-than
  ack-imp (m,n) returns r
  assumes m≥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)
  }
)
supply nat-distribs[simp]
by vcg-cs

6.6.4 McCarthy’s 91 Function

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

recursive-spec p91(y) assumes True ensures if 100<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)
  }
)
apply vcg-cs
apply (auto split: if-splits)
done
6.6.5 Odd/Even

recursive-spec
odd-imp (a) returns b
assumes True
ensures b\neq 0 \iff odd a_0
variant |a|
defines :
\begin{align*}
& \text{if } (a = 0) b = 0 \\
& \text{else if } (a < 0) b = \text{rec even-imp } (a+1) \\
& \text{else } b = \text{rec even-imp } (a-1)
\end{align*}
and
even-imp (a) returns b
assumes True
ensures b\neq 0 \iff even a_0
variant |a|
defines :
\begin{align*}
& \text{if } (a = 0) b = 1 \\
& \text{else if } (a < 0) b = \text{rec odd-imp } (a+1) \\
& \text{else } b = \text{rec odd-imp } (a-1)
\end{align*}
apply vcg
apply auto
done

thm even-imp-spec

6.6.6 Pandya and Joseph’s Product Producers

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

recursive-spec relation (measure nat <*lex*> less-than)
product () assumes True ensures \( GZ = GZ_0 + GX_0 \ast GY_0 \)
variant (|GY|,1::nat)
for GX GY GZ
defines
e = even-imp (GY);
if (e\neq 0) rec evenproduct() else rec oddproduct()
and
oddproduct() assumes (odd GY) ensures \( GZ = GZ_0 + GX_0 \ast GY_0 \)
variant (|GY|,0::nat)
for GX GY GZ
defines
if (GY<0) {
  GY = GY + 1;
}
\[ GZ = GZ - GX \]
\}
\else \{
\}
\}
\)
\)
\}
\)
\}
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
{0..<ptr₀} ∧ ptr = ptr₀ + 1

for stack[]
defines (stack[ptr] = x; ptr = ptr + 1)
by vcg-cs (auto simp: fun-upd-image)

program-spec get-succs
assumes j ≤ stop ∧ stop = a (j - 1) ∧ 0 ≤ i
ensures
stack · {0..<i} = {x. (j₀ - 1, x) ∈ Edges a ∧ x ∉ set-of visited}
∪ stack₀ · {0..<i₀}
∧ i ≥ i₀
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})
∧ i ≥ i₀ ∧ i ≤ j ∧ j₀ ≤ j
@
variant ((stop - j)):

{ succ = a[j];
  is-elem = bv-elem(succ,visited);
  if (is-elem == 0) {
    (stack, i) = push (succ, stack, i)
  };
  j = j + 1
}

by vcg-cs (auto simp: intvs-incr-h Edges-def succs-def)

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

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

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

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

**Lemma**: reachability-invariant:

- Assumes reachable: \((s, x) \in (\text{Edges } a)^*\)
  and closed: \(\forall v \in \text{visited}. \text{Edges } a \upharpoonright \{v\} \subseteq \text{visited}\)
  and start: \(s \in \text{visited}\)
- Shows \(x \in \text{visited}\)

**Using** reachable start closed by induction auto

**Program-spec** (partial) dfs

- Assumes \(0 \leq x \land 0 \leq s\)
- Ensures \(b = 1 \iff x \in (\text{Edges } a)^* \upharpoonright \{s\}\)
  \(b = 0\);
- Clear stack[];
- \(i = \text{stack-init}()\);
- \((\text{stack}, i) = \text{push} (s, \text{stack}, i)\);
- Clear visited[];
- While \((b == 0 \land i \neq 0)\)

  @invariant \(0 \leq i \land (s \in \text{stack} \upharpoonright \{0..<i\} \lor s \in \text{set-of visited}) \land \)

  \((b = 0 \lor b = 1) \land (\)
  - if \(b = 0\) then
    - \(\text{stack} \upharpoonright \{0..<i\} \subseteq (\text{Edges } a)^* \upharpoonright \{s\}\)
    - \(\land (\forall v \in \text{set-of visited}. (\text{Edges } a)^* \upharpoonright \{v\} \subseteq \text{set-of visited} \cup \text{stack} \)
      \(\upharpoonright \{0..<i\}\))
    - \((x \notin \text{set-of visited})\)
    - else \(x \in (\text{Edges } a)^* \upharpoonright \{s\}\)

  )

  \((\text{next}, i) = \text{pop}(\text{stack}, i)\); — Take the top most element from the stack.
  \(\text{visited} = \text{bv-insert}(\text{next}, \text{visited})\); — Mark it as visited,
  if \((\text{next} == x)\) {
    \(b = 1\) — If it is the target, we are done.
  } else {
    — Else, put its successors on the stack if they are not yet visited.
    \(\text{stop} = a[\text{next}]\);
    \(j = \text{next} + 1;\)
    if \((j \leq \text{stop})\) {
      inline get-succs
    }
  }

apply vcg-cs

subgoal by (auto simp: set-of-def)

subgoal using intvs-lower-incr by (auto simp: Edges-empty)

subgoal by auto (fastforce simp: set-of-def dest!: reachability-invariant)
done

78
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`: 

\[ s \subseteq \text{insert } x s \iff x \not\in s \]

by auto

**program-spec dfs1**

- **assumes** `0 \leq x \land 0 \leq s \land start = s \land edges = a`
- **ensures** `b = 1 \iff x \in (Edges a)^* " \{s\}`

**for** `visited[]` **defines**

\[
\begin{align*}
& b = 0 ; \\
& \begin{array}{l}
\text{— } i \text{ will point to the next free space in the stack (i.e. it is the size of the stack)} \\
\text{— Initially, we put} \ s \text{ on the stack.} \\
\text{stack}[0] = s ; \\
\text{visited} = \text{bv-init} () ; \\
\text{while} \ (b == 0 \land i \neq 0) \\
\text{@invariant} : \\
\begin{array}{l}
0 \leq i \land (s \in \text{stack} \ \{0..<i\} \lor s \in \text{set-of visited}) \land (b = 0 \lor b = 1) \\
\land \text{set-of visited} \subseteq (Edges edges)^* " \{start\} \land ( \\
\text{if } b = 0 \text{ then} \\
\text{stack}' \ {0..<i} \subseteq (Edges a)^* " \{s\} \\
\land (\forall v \in \text{set-of visited}. \ (Edges a)^* " \{v\} \subseteq \text{set-of visited} \cup \text{stack} ' \ {0..<i} ) \\
\text{else } x \in (Edges a)^* " \{s\} \\
\text{)}
\end{array}
\end{array}
\end{align*}
\]

@relation `finite-psupset ((Edges edges)^* " \{start\}) <*lex* > less-than`

@variant `(set-of visited, nat i)`

\[
\begin{align*}
& \text{— Take the top most element from the stack.} \\
& \text{next, i} = \text{pop}(\text{stack, i}) ; \\
& \text{if} \ (\text{next} == x) \{ \\
& \text{— If it is the target, we are done.} \\
& \text{visited} = \text{bv-insert}(\text{next, visited}) ; \\
& b = 1 \\
& \} \text{ else } \{ \\
& \text{is-elem} = \text{bv-elem}(\text{next, visited}) ; \\
& \text{if} \ (\text{is-elem} == 0) \{ \\
& \text{visited} = \text{bv-insert}(\text{next, visited}) ; \\
& \text{— Else, put its successors on the stack if they are not yet visited.}
\end{align*}
\]

79
\texttt{stop} = a[\texttt{next}];
\texttt{j} = \texttt{next} + 1;
\texttt{if} (\texttt{j} \leq \texttt{stop}) {
    \texttt{inline get-succs}
}

\texttt{apply \texttt{vcg-cs}}
\texttt{subgoal by auto}
\texttt{subgoal by (auto simp add: image-constant-conv)}
\texttt{subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)}
\texttt{subgoal by (auto simp: set-of-def)}
\texttt{subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)}
\texttt{subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)}
\texttt{subgoal by (auto simp: Edges-empty)}
\texttt{subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)}
\texttt{subgoal by (auto simp: set-of-def)}
\texttt{subgoal by auto (fastforce simp: set-of-def dest!: reachability-invariant)}
\texttt{done}

\texttt{end}

\texttt{end}