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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.3 Structure Types</td>
<td>32</td>
</tr>
<tr>
<td>4.6.4 Function Calls as Expressions</td>
<td>32</td>
</tr>
<tr>
<td>4.6.5 Ghost Variables</td>
<td>33</td>
</tr>
<tr>
<td>4.6.6 Concurrency</td>
<td>33</td>
</tr>
<tr>
<td>4.6.7 Pointers and Memory</td>
<td>33</td>
</tr>
<tr>
<td>5 Introduction to IMP2-VCG, based on IMP</td>
<td>33</td>
</tr>
<tr>
<td>5.1 Fancy Syntax</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Operators and Arrays</td>
<td>34</td>
</tr>
<tr>
<td>5.3 Local and Global Variables</td>
<td>35</td>
</tr>
<tr>
<td>5.3.1 Parameter Passing</td>
<td>35</td>
</tr>
<tr>
<td>5.4 Recursive procedures</td>
<td>35</td>
</tr>
<tr>
<td>5.4.1 Procedure Scope</td>
<td>35</td>
</tr>
<tr>
<td>5.4.2 Syntactic sugar for procedure call with parameters</td>
<td>35</td>
</tr>
<tr>
<td>5.5 More Readable VCs</td>
<td>35</td>
</tr>
<tr>
<td>5.6 Specification Commands</td>
<td>36</td>
</tr>
<tr>
<td>6 Examples</td>
<td>37</td>
</tr>
<tr>
<td>6.1 Common Loop Patterns</td>
<td>37</td>
</tr>
<tr>
<td>6.1.1 Count Up</td>
<td>37</td>
</tr>
<tr>
<td>6.1.2 Count down</td>
<td>40</td>
</tr>
<tr>
<td>6.1.3 Approximate from Below</td>
<td>40</td>
</tr>
<tr>
<td>6.1.4 Bisection</td>
<td>41</td>
</tr>
<tr>
<td>6.2 Debugging</td>
<td>42</td>
</tr>
<tr>
<td>6.2.1 Testing Programs</td>
<td>42</td>
</tr>
<tr>
<td>6.3 More Numeric Algorithms</td>
<td>42</td>
</tr>
<tr>
<td>6.3.1 Euclid's Algorithm (with subtraction)</td>
<td>42</td>
</tr>
<tr>
<td>6.3.2 Euclid's Algorithm (with mod)</td>
<td>42</td>
</tr>
<tr>
<td>6.3.3 Extended Euclid's Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>6.3.4 Exponentiation by Squaring</td>
<td>45</td>
</tr>
<tr>
<td>6.3.5 Power-Tower of 2s</td>
<td>45</td>
</tr>
<tr>
<td>6.4 Array Algorithms</td>
<td>46</td>
</tr>
<tr>
<td>6.4.1 Summation</td>
<td>46</td>
</tr>
<tr>
<td>6.4.2 Finding Least Index of Element</td>
<td>47</td>
</tr>
<tr>
<td>6.4.3 Check for Sortedness</td>
<td>47</td>
</tr>
<tr>
<td>6.4.4 Find Equilibrium Index</td>
<td>47</td>
</tr>
<tr>
<td>6.4.5 Rotate Right</td>
<td>48</td>
</tr>
<tr>
<td>6.4.6 Binary Search, Leftmost Element</td>
<td>49</td>
</tr>
<tr>
<td>6.4.7 Naive String Search</td>
<td>49</td>
</tr>
<tr>
<td>6.4.8 Insertion Sort</td>
<td>52</td>
</tr>
<tr>
<td>6.4.9 Quicksort</td>
<td>53</td>
</tr>
<tr>
<td>6.5 Data Refinement</td>
<td>55</td>
</tr>
<tr>
<td>6.5.1 Filtering</td>
<td>55</td>
</tr>
<tr>
<td>6.5.2 Merge Two Sorted Lists</td>
<td>56</td>
</tr>
<tr>
<td>6.5.3 Remove Duplicates from Array, using Bitvector</td>
<td>58</td>
</tr>
<tr>
<td>6.6 Recursion</td>
<td>60</td>
</tr>
<tr>
<td>6.6.1 Recursive Fibonacci</td>
<td>60</td>
</tr>
<tr>
<td>6.6.2 Homeier’s Cycling Termination</td>
<td>61</td>
</tr>
<tr>
<td>6.6.3 Ackermann</td>
<td>61</td>
</tr>
</tbody>
</table>
1 Abstract Syntax of IMP2

theory Syntax
imports Main
begin

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

1.1 Primitives

Variable and procedure names are strings.

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

The variable names are partitioned into local and global variables.

fun is-global :: vname ⇒ bool where
  is-global [] ←→ True
  | is-global (CHR "G"#-) ←→ True
  | is-global - ←→ False

abbreviation is-local a ≡ ¬is-global a

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

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

type-synonym pval = int
type-synonym val = int ⇒ pval

1.2 Arithmetic Expressions

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

datatype aexp =
  N int
Vidx vname aexp
| Unop int ⇒ int aexp
| Binop int ⇒ int ⇒ int aexp aexp

1.3 Boolean Expressions

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

```
datatype bexp =
  Bc bool
  | Not bexp
  | BBinop bool ⇒ bool ⇒ bool bexp bexp
  | Cmpop int ⇒ int ⇒ bool aexp aexp
```

1.4 Commands

The commands can roughly be put into five categories:

**Skip** The no-op command

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

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

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

```
datatype com =
  SKIP — No-op
```
— Assignment
| AssignIdx vname aexp aexp — Assign to index in array
| ArrayCpy vname vname — Copy whole array
| ArrayClear vname — Clear array
| Assign-Locals vname ⇒ val — Internal: Assign all local variables simultaneously

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

— Procedure
| PCall pname — Procedure call
| PScope pname → com com — Procedure scope

1.4.1 Minimal Concrete Syntax

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

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

1.5 Program

type-synonym program = pname → com

1.6 Default Array Index

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

abbreviation V x ≡ Vidx x (N 0)
abbreviation Assign (\(-::=-\([1000, 61]\] 61)\)
where x := a ≡ (x[N 0] := a)
2 Semantics of IMP

theory Semantics
imports Syntax HOL Eisbach.Eisbach-Tools
begin

2.1 State

The state maps variable names to values

type-synonym state = vname ⇒ val

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

definition null-state (<>)
where
null-state ≡ λ x. λ xi. 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 = (if is-local n then s n else t n)

We prove some basic facts.

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

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

lemma combine-collapse: <s|s> = s ⟨proof⟩

lemma combine-nest:
<s|t> = <s|t>
<s|t'> | t> = <s|t'> | t>
⟨proof⟩

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

lemma combine-upd:
is-local \( x \Rightarrow <s|t>(x:=v) = <s(x:=v)|t> \)
is-global \( x \Rightarrow <s|t>(x:=v) = <s|t(x:=v)> \)

\( \langle \text{proof} \rangle \)

lemma combine-cases[cases type]:
\begin{align*}
on t &\quad \text{obtains } l \ g \text{ where } s = <l|g>
\end{align*}
\( \langle \text{proof} \rangle \)
end

2.2 Arithmetic Expressions

The evaluation of arithmetic expressions is straightforward.

fun aval :: aexp ⇒ state ⇒ pval where
\begin{align*}
\text{aval} (N n) s &= n \\
\text{aval} (Vidx x i) s &= s x (\text{aval} i s) \\
\text{aval} (Unop f a_1) s &= f (\text{aval} a_1 s) \\
\text{aval} (Binop f a_1 a_2) s &= f (\text{aval} a_1 s) (\text{aval} a_2 s)
\end{align*}

2.3 Boolean Expressions

The evaluation of Boolean expressions is straightforward.

fun bval :: bexp ⇒ state ⇒ bool where
\begin{align*}
\text{bval} (Bc v) s &= v \\
\text{bval} (Not b) s &= (\neg \text{bval} b s) \\
\text{bval} (BBinop f b_1 b_2) s &= f (\text{bval} b_1 s) (\text{bval} b_2 s) \\
\text{bval} (Cmpop f a_1 a_2) s &= f (\text{aval} a_1 s) (\text{aval} a_2 s)
\end{align*}

2.4 Big-Step Semantics

The big-step semantics is a relation from commands and start states to end states, such that there is a terminating execution. If there is no such execution, no end state will be related to the command and start state. This either means that the program does not terminate, or gets stuck because it tries to call an undefined procedure.

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

inductive big-step :: program ⇒ com × state ⇒ state ⇒ bool
\begin{align*}
\text{No-Op} &\quad - \Rightarrow - [1000,55,55] 55 \\
\text{AssignIdx} &\quad \pi:(x[i] ::= a,s) \Rightarrow s(x := (s x)(\text{aval} i s := \text{aval} a s))
\end{align*}
| ArrayInitE: $\pi(x[]) ::= y, s) \Rightarrow s(x := s y)$ |
| ArrayClear: $\pi(\text{CLEAR} x[], s) \Rightarrow s(x := \lambda. 0)$ |
| Assign-Locals: $\pi(\text{Assign-Locals} l, s) \Rightarrow <l|s)$

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

— Procedure commands
| PCall: $[\pi p = \text{Some } c; \pi(c, s) \Rightarrow t] \Rightarrow \pi(\text{PCall } p, s) \Rightarrow t$ |
| PScope: $[\pi'(c, s) \Rightarrow t] \Rightarrow \pi(\text{PScope } p', c) \Rightarrow t$ |

2.4.1 Proof Automation

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

\texttt{declare \text{big-step.intros \ intro}}

\texttt{lemmas \text{big-step-induct[\text{induct set}] = \text{big-step.induct[\text{split-format\{complete\}]}}}}

\texttt{inductive-simps \text{Skip-simp: } \pi(\text{SKIP}, s) \Rightarrow t}$
\texttt{inductive-simps \text{AssignIdx-simp: } \pi(x[], s) := a, s) \Rightarrow t}$
\texttt{inductive-simps \text{ArrayCpy-simp: } \pi(x[]) := g, s) \Rightarrow t}$
\texttt{inductive-simps \text{ArrayInit-simp: } \pi(\text{CLEAR} x[], s) \Rightarrow t}$
\texttt{inductive-simps \text{AssignLocals-simp: } \pi(\text{Assign-Locals} l, s) \Rightarrow t}$

\texttt{inductive-simps \text{Seq-simp: } \pi(c_1; c_2, s_1) \Rightarrow s_2}$
\texttt{inductive-simps \text{If-simp: } \pi(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \Rightarrow t}$
\texttt{inductive-simps \text{Scope-simp: } \pi(\text{SCOPE} c, s) \Rightarrow t}$
\texttt{inductive-simps \text{PCall-simp: } \pi(\text{PCall } p, s) \Rightarrow t}$
\texttt{inductive-simps \text{PScope-simp: } \pi(\text{PScope } p', p, s) \Rightarrow t}$

\texttt{lemmas \text{big-step-simps =}}
\texttt{Skip-simp \text{AssignIdx-simp ArrayCpy-simp ArrayInit-simp}}
\texttt{Seq-simp \text{If-simp Scope-simp PCall-simp PScope-simp}}

\texttt{inductive-cases \text{SkipE[elim!]}: \pi(\text{SKIP}, s) \Rightarrow t}$
\texttt{inductive-cases \text{AssignIdxE[elim!]}: \pi(x[], s) := a, s) \Rightarrow t}$
\texttt{inductive-cases \text{ArrayCpyE[elim!]}: \pi(x[], s) := a, s) \Rightarrow t}$
\texttt{inductive-cases \text{ArrayInitE[elim!]}: \pi(c, s) \Rightarrow \text{true}}$
\texttt{inductive-cases \text{AssignLocalsE[elim!]}: \pi(\text{Assign-Locals} l, s) \Rightarrow t}$
2.4.2 Automatic Derivation

\textbf{lemma} Assign': \( s' = s(x := (s x)(aval i s := aval a s)) \Rightarrow \pi:(x[i]\leftarrow a, s) \Rightarrow s' \langle \text{proof}\rangle \)

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

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

\textbf{lemma} Scope': \( s_1 = <\_\_|s> \Rightarrow \pi:(c,s_1) \Rightarrow t \Rightarrow t' = <s|t> \Rightarrow \pi:(\text{Scope } c,s) \Rightarrow t' \langle \text{proof}\rangle \)

\textbf{named-theorems} deriv-unfolds \( \langle \text{Unfold rules before derivations}\rangle \)

\textbf{method} bs-simp = simp add: combine-nest combine-upd combine-query fun-upd-same fun-upd-other del: fun-upd-apply

\textbf{method} big-step' =
\hspace{1em} \text{rule Skip } \text{Seq } \text{PScope}
\hspace{1em} | \text{ (rule Assign' ArrayCpy' ArrayClear', (bs-simp:fail))}
\hspace{1em} | \text{ (rule IfTrue IfFalse WhileTrue WhileFalse PCall Scope'), (bs-simp:fail)}
\hspace{1em} | \text{ unfold deriv-unfolds}
\hspace{1em} | \text{ (bs-simp: fail)}

\textbf{method} big-step =
\hspace{1em} \text{rule Skip}
\hspace{1em} | \text{ rule Seq, (big-step:fail), (big-step:fail)}
\hspace{1em} | \text{ rule PScope, (big-step:fail)}
\hspace{1em} | \text{ (rule Assign' ArrayCpy' ArrayClear', (bs-simp:fail))}
\hspace{1em} | \text{ (rule IfTrue IfFalse, (bs-simp:fail), (big-step:fail))}
\hspace{1em} | \text{ rule WhileTrue, (bs-simp:fail), (big-step:fail), (big-step:fail)}
\hspace{1em} | \text{ rule WhileFalse, (bs-simp:fail)}
\hspace{1em} | \text{ rule PCall, (bs-simp:fail), (big-step:fail)}
\hspace{1em} | \text{ (rule Scope', (bs-simp:fail), (big-step:fail), (bs-simp:fail))}
\hspace{1em} | \text{ unfold deriv-unfolds, big-step}

\textbf{schematic-goal} Map.empty: ("a" := N 1;;
\hspace{1em} WHILE Cmpop (\lambda x. y < x) (V "n") (N 0) DO (}
\[ \text{"a" ::= Binop (+) (V "a") (V "a")}; \]
\[ \text{"n" ::= Binop (-) (V "n") (N 1) } \]
\[ \text{,<"n":=(λ. 5)>) ⇒ s} \]

\[ \langle \text{proof} \rangle \]

\section{2.5 Command Equivalence}

Two commands are equivalent if they have the same semantics.

\textbf{definition}
\[ \text{equiv-c :: com ⇒ com ⇒ bool (infix \sim 50) where} \]
\[ c \sim c' \equiv (\forall s t. \pi:(c,s) ⇒ t = \pi:(c',s) ⇒ t) \]

\textbf{lemma equivI[intro]}:
\[ \forall s t. \pi:(c,s) ⇒ t \implies \pi:(c',s) ⇒ t \]
\[ \implies c \sim c' \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma equivD[dest]}:
\[ c \sim c' \implies \pi:(c,s) ⇒ t \iff \pi:(c',s) ⇒ t \]
\[ \langle \text{proof} \rangle \]

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

\textbf{lemma equiv-refl[simp, intro]}: \[ c \sim c \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma equiv-sym[trans]}: \[ (c \sim c') \implies (c' \sim c) \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma equiv-trans[trans]}: \[ c \sim c' \implies c' \sim c'' \implies c \sim c'' \]
\[ \langle \text{proof} \rangle \]

\section{2.5.1 Basic Equivalences}

\textbf{lemma while-unfold}:
\[ (\text{WHILE } b \text{ DO } c) \sim (\text{IF } b \text{ THEN } c;\; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP}) \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma triv-if}:
\[ (\text{IF } b \text{ THEN } c \text{ ELSE } c) \sim c \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma commute-if}:
\[ (\text{IF } b1 \text{ THEN } (\text{IF } b2 \text{ THEN } c11 \text{ ELSE } c12) \text{ ELSE } c2) \sim (\text{IF } b2 \text{ THEN } (\text{IF } b1 \text{ THEN } c11 \text{ ELSE } c2) \text{ ELSE } (\text{IF } b1 \text{ THEN } c12 \text{ ELSE } c2)) \]
\[ \langle \text{proof} \rangle \]

\textbf{lemma sim-while-cong-aux}:
\[ \pi: (\text{WHILE } b \text{ DO } c, s) \Rightarrow t \quad \text{bval } b = \text{bval } b'; c \sim c' \implies \pi: (\text{WHILE } b' \text{ DO } c', s) \Rightarrow t \]  
(proof)

**lemma sim-while-cong:** bval b = bval b' \implies c \sim c' \implies WHILE b DO c \sim WHILE b' DO c'  
(proof)

### 2.6 Execution is Deterministic

This proof is automatic.

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

### 2.7 Small-Step Semantics

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

Moreover, the step function is explicitly partial, returning \textit{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 \textit{SKIP}, then the sequential composition is reduced to the second command.

A while command is reduced by unfolding the loop once.

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

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

**fun small-step :: program ⇒ com × state → com × state where**  
\[ \text{small-step } \pi (x[i]):=a, s) = \text{Some } (\text{SKIP}, s(x := (s x)(\text{aval } i \ s := \text{aval } a s))) \]  
| small-step \( \pi (x[i]):=y, s) = \text{Some } (\text{SKIP}, s(x := s y)) \)  
| small-step \( \pi (\text{CLEAR } x[i], s) = \text{Some } (\text{SKIP}, s(x := (\lambda\cdot 0))) \)  
| small-step \( \pi (\text{Assign-Locals } l, s) = \text{Some } (\text{SKIP},<l|s>) \)  
| small-step \( \pi (\text{SKIP};c, s) = \text{Some } (c, s) \)  
| small-step \( \pi (c_1; c_2, s) = (\text{case small-step } \pi (c_1, s) \text{ of Some } (c_1', s') \Rightarrow \text{Some } (c_1'; c_2, s') | - \Rightarrow \text{None} \)  
| small-step \( \pi (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) = \text{Some } (\text{if bval } b s \text{ then } (c_1, s) \text{ else } (c_2, s)) \)  
| small-step \( \pi (\text{SCOPE } c, s) = \text{Some } (c; \text{Assign-Locals } s, <<>|s>) \)
We define the reflexive transitive closure of the step function.

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

\[\begin{align*}
[\text{simp}]: & \quad \text{small-steps } \pi \text{ cs} (\text{Some cs}) \\
[\text{small-step } \pi (\text{WHILE } b \text{ DO } c, s) = \text{Some (IF } b \text{ THEN } c;\text{WHILE } b \text{ DO } c \text{ ELSE } \text{SKIP, s}) \Rightarrow \text{None}] \\
[\text{small-step } \pi (\text{PCall } p, s) = \text{Some (SKIP, s)}] \\
[\text{small-step } \pi (\text{PScope } \pi' \text{ SKIP, s}) = \text{Some (SKIP, s)}] \\
[\text{small-step } \pi (\text{PScope } \pi' \text{ c, s}) = (\text{case small-step } \pi'(c, s) \text{ of Some } (c', s') \Rightarrow \text{Some (PScope } \pi' \text{ c', s') } | - \Rightarrow \text{None}] \\
[\text{small-step } \pi (\text{SKIP, s}) = \text{Some (SKIP, s)}]
\end{align*}\]

\[\text{lemma small-steps-append: small-steps } \pi \text{ cs}_1 (\text{Some cs}_2) \Rightarrow \text{small-steps } \pi \text{ cs}_2 \text{ cs}_3 \Rightarrow \text{small-steps } \pi \text{ cs}_1 \text{ cs}_3\]

\[\langle \text{proof} \rangle\]

### 2.7.1 Equivalence to Big-Step Semantics

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

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

\[\text{lemma small-big-append: small-step } \pi \text{ cs}_1 = \text{Some cs}_2 \Rightarrow \pi: \text{cs}_2 \Rightarrow s_3 \Rightarrow \pi: \text{cs}_1 \Rightarrow s_3\]

\[\langle \text{proof} \rangle\]

\[\text{lemma smalls-big-append: small-steps } \pi \text{ cs}_1 (\text{Some cs}_2) \Rightarrow \pi: \text{cs}_2 \Rightarrow s_3 \Rightarrow \pi: \text{cs}_1 \Rightarrow s_3\]

\[\langle \text{proof} \rangle\]

\[\text{lemma small-imp-big:}\]

\[\text{assumes small-steps } \pi \text{ cs}_1 (\text{Some (SKIP, s)}_2)\]

\[\text{shows } \pi: \text{cs}_1 \Rightarrow s_2\]

\[\langle \text{proof} \rangle\]

\[\text{lemma small-steps-skip-term[simp]: small-steps } \pi (\text{SKIP, s}) \text{ cs'} \leftrightarrow cs' = \text{Some (SKIP, s)}\]

\[\langle \text{proof} \rangle\]

\[\text{lemma small-seq: } [c \neq \text{SKIP}; \text{small-step } \pi (c, s) = \text{Some (c', s')}] \Rightarrow \text{small-step } \pi (c;\text{cx, s}) = \text{Some (c';cx, s')}\]

\[\langle \text{proof} \rangle\]
lemma smalls-seq:\[\text{small-steps } \pi (c, s) (\text{Some } (c', s')) \implies \text{small-steps } \pi (c \mathrel{;;} c', s, s')\]
⟨proof⟩

lemma small-pscope:
\[\llbracket c \neq \text{SKIP}; \text{small-step } \pi' (c, s) = \text{Some } (c', s') \rrbracket \implies \text{small-step } \pi (\text{PScope } \pi' c, s) = \text{Some } (\text{PScope } \pi' c', s')\]
⟨proof⟩

lemma smalls-pscope:
\text{small-steps } \pi' (c, s) (\text{Some } (c', s')) \implies \text{small-steps } \pi (\text{PScope } \pi' c, s) (\text{Some } (\text{PScope } \pi' c', s'))
⟨proof⟩

lemma big-imp-small:
assumes \(\pi: cs \Rightarrow t\)
shows \text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))
⟨proof⟩

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

theorem big-eq-small: \(\pi: cs \Rightarrow t \iff \text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))\)
⟨proof⟩

lemma small-steps-determ:
assumes \text{small-steps } \pi cs None
shows \neg \text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))
⟨proof⟩

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

corollary small-imp-big-fail:
assumes \text{small-steps } \pi cs None
shows \(\nexists t. \pi: cs \Rightarrow t\)
⟨proof⟩

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** \( \text{wp} \) \( c \) \( Q \) \( s \) \( \equiv \exists t. \pi: (c,s) \Rightarrow t \land Q t \)

— 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** \( \text{wlp} \) \( c \) \( Q \) \( s \) \( \equiv \forall t. \pi: (c,s) \Rightarrow t \rightarrow Q t \)

2.8.1 Basic Properties

**Context**

**Notes** \( \text{abs-def,simp} = \text{wp-def wlp-def} \)

**Begin**

**Lemma** \( \text{wp-imp-wlp} \): \( \text{wp} \) \( c \) \( Q \) \( s \) \( \Rightarrow \text{wlp} \) \( c \) \( Q \) \( s \)  

**Proof**

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

**Proof**

**Lemma** \( \text{wp-equiv} \): \( c \) \( \sim \) \( c' \) \( \Rightarrow \) \( \text{wp} \) \( c \) \( c' \) \( \Rightarrow \) \( \text{wp} \) \( c' \) \( \Rightarrow \) \( \text{wp} \) \( c \) \( \Rightarrow \) \( \text{wp} \) \( c' \)  

**Proof**

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

**Proof**

**Lemma** \( \text{wlp-equiv} \): \( c \) \( \sim \) \( c' \) \( \Rightarrow \) \( \text{wlp} \) \( c \) \( c' \) \( \Rightarrow \) \( \text{wlp} \) \( c' \) \( \Rightarrow \) \( \text{wlp} \) \( c \) \( \Rightarrow \) \( \text{wlp} \) \( c' \)  

**Proof**

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

**Proof**

2.8.2 Unfold Rules

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

**Proof**

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

**Proof**

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

**Proof**

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

**Proof**

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

**Proof**

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

**Proof**

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

**Proof**

**Lemma** \( \text{wp-scope-eq} \): \( \text{wp} \) \( \text{SCOPE} c \) \( Q s \) \( = \) \( \text{wp} \) \( \text{SCOPE} \) \( c \) \( \text{Q s} \) \( = \) \( \text{wp} \) \( \text{SCOPE} \) \( c \) \( \text{Q s} \)  

**Proof**

**Lemma** \( \text{wp-pcall-eq} \): \( \text{wp} \) \( \text{PCall} p \) \( Q s \) \( = \) \( \text{wp} \) \( \text{PCall} \) \( p \) \( Q s \) \( = \) \( \text{wp} \) \( \text{PCall} \) \( p \) \( Q s \)  

**Proof**

15
lemmas \( wp\text{-eq} = wp\text{-skip-eq} \) wp\text{-assign-idx-eq} wp\text{-arraycpy-eq} wp\text{-arrayinit-eq} wp\text{-assign-locals-eq} wp\text{-seq-eq} wp\text{-scope-eq} \\
lemmas \( wp\text{-eq}' = wp\text{-eq} \) wp\text{-if-eq} \\

lemma \( wlp\text{-skip-eq}: wlp \ SKIP \ Q \ s = Q \ s \langle \text{proof} \rangle \)
lemma \( wlp\text{-assign-idx-eq}: wlp \ (x[i]::=a) \ Q s = Q \ (s(x:=(s \ x)(aval i s := aval a \ s))) \langle \text{proof} \rangle \)
lemma \( wlp\text{-arraycpy-eq}: wlp \ (x[]::=a) \ Q s = Q \ (s(x:=s \ a)) \langle \text{proof} \rangle \)
lemma \( wlp\text{-assign-locals-eq}: wlp \ (Assign-Locals \ l) \ Q s = Q \langle l | s \rangle \langle \text{proof} \rangle \)
lemma \( wlp\text{-seq-eq}: wlp \ (c_1;;c_2) \ Q s = wlp c_1 \ (wlp c_2 \ Q) \ s \langle \text{proof} \rangle \)
lemma \( wlp\text{-if-eq}: wlp \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ Q s = (if \ bval \ b \ s \ then \ wlp \ c_1 \ Q \ s \ else \ wlp \ c_2 \ Q \ s) \langle \text{proof} \rangle \)
lemma \( wlp\text{-while-unfold}: wlp \ (WHILE \ b \ DO \ c) \ Q s = (if \ bval \ b \ s \ then \ wlp \ c \ (wlp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \ Q \ s) \langle \text{proof} \rangle \)
lemma \( wp\text{-while-unfold}: wp \ (WHILE \ b \ DO \ c) \ Q s = (if \ bval \ b \ s \ then \ wp \ c \ (wp \ (WHILE \ b \ DO \ c) \ Q) \ s \ else \ Q \ s) \langle \text{proof} \rangle \)

lemmas \( wp\text{-eq} = wlp\text{-skip-eq} \) wlp\text{-assign-idx-eq} wlp\text{-arraycpy-eq} wlp\text{-arrayinit-eq} wlp\text{-assign-locals-eq} wlp\text{-seq-eq} wlp\text{-scope-eq} \\
lemmas \( wp\text{-eq}' = wp\text{-eq} \) wp\text{-if-eq} \\
end \\

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

end — Context fixing program

Unfold rules for procedure scope
lemma \( wp\text{-pscope-eq}: wp \ \pi \ (PScope \ \pi' \ c) \ Q s = wp \ \pi' \ (c) \ Q s \langle \text{proof} \rangle \)
lemma \( wlp\text{-pscope-eq}: wlp \ \pi \ (PScope \ \pi' \ c) \ Q s = wlp \ \pi' \ (c) \ Q s \langle \text{proof} \rangle \)

2.8.3 Weakest precondition and Program Equivalence

The following three statements are equivalent:
1. The commands $c$ and $c'$ are equivalent
2. The weakest preconditions are equivalent, for all procedure environments
3. The weakest liberal preconditions are equivalent, for all procedure environments

\textbf{Lemma wp-equiv-iff}: $(\forall \pi. \wp \pi c = \wp \pi c') \iff c \sim c'$  
\textlangle proof \textrangle

\textbf{Lemma wlp-equiv-iff}: $(\forall \pi. \wlp \pi c = \wlp \pi c') \iff c \sim c'$  
\textlangle proof \textrangle

\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 sim-while-wp-aux}:  
\texttt{assumes bval b = bval b'} \hfill  
\texttt{assumes wp \pi c = wp \pi c'} \hfill  
\texttt{assumes \pi: (WHILE b DO c, s) \Rightarrow t} \hfill  
\texttt{shows \pi: (WHILE b' DO c', s) \Rightarrow t} \hfill  
\textlangle proof \textrangle

\textbf{Lemma sim-while-wp}: $bval b = bval b' \implies \wp \pi c = \wp \pi c' \implies \wp \pi (\text{WHILE } b \text{ DO } c) = \wp \pi (\text{WHILE } b' \text{ DO } c')$  
\textlangle proof \textrangle

The same lemma for weakest liberal preconditions.

\textbf{Lemma sim-while-wlp-aux}:  
\texttt{assumes bval b = bval b'} \hfill  
\texttt{assumes wlp \pi c = wlp \pi c'} \hfill  
\texttt{assumes \pi: (WHILE b DO c, s) \Rightarrow t} \hfill  
\texttt{shows \pi: (WHILE b' DO c', s) \Rightarrow t} \hfill  
\textlangle proof \textrangle

\textbf{Lemma sim-while-wlp}: $bval b = bval b' \implies wlp \pi c = wlp \pi c' \implies wlp \pi (\text{WHILE } b \text{ DO } c) = wlp \pi (\text{WHILE } b' \text{ DO } c')$  
\textlangle proof \textrangle

\subsection{2.9 Invariants for While-Loops}

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

lemma wlp-whileI':
assumes INIT: \( I s_0 \)
assumes STEP: \( \forall s. \ I s \implies (\text{if } bval b s \text{ then } \text{wlp } \pi c I s \text{ else } Q s) \)
shows \( \text{wlp } \pi (\text{WHILE } b \text{ DO } c) Q s_0 \)
⟨proof⟩

lemma
assumes INIT: \( I s_0 \)
assumes STEP: \( \forall s. \ I s \implies (\text{if } bval b s \text{ then } \text{wlp } \pi c I s \text{ else } Q s) \)
shows \( \text{wlp } \pi (\text{WHILE } b \text{ DO } c) Q s_0 \)
⟨proof⟩

2.9.2 Total Correctness

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

lemma wp-whileI':
assumes WF: \( \text{wf } R \)
assumes INIT: \( I s_0 \)
assumes STEP: \( \forall s. \ I s \implies (\text{if } bval b s \text{ then } \text{wp } \pi c (\lambda s'. \ I s' \land (s',s) \in R) \text{ s else } Q s) \)
shows \( \text{wp } \pi (\text{WHILE } b \text{ DO } c) Q s_0 \)
⟨proof⟩

lemma
assumes WF: \( \text{wf } R \)
assumes INIT: \( I s_0 \)
assumes STEP: \( \forall s. \ I s \implies (\text{if } bval b s \text{ then } \text{wp } \pi c (\lambda s'. \ I s' \land (s',s) \in R) \text{ s else } Q s) \)
shows \( \text{wp } \pi (\text{WHILE } b \text{ DO } c) Q s_0 \)
⟨proof⟩

2.9.3 Standard Forms of While Rules

lemma wlp-whileI:
assumes INIT: \( I s_0 \)
assumes STEP: \( \forall s. \ I s \implies \text{wp } \pi c I s \)
assumes FINAL: \( \forall s. \ I s \implies \text{wp } \pi c Q s \)
shows \( \text{wlp } \pi (\text{WHILE } b \text{ DO } c) Q s_0 \)
⟨proof⟩

lemma wp-whileI:
assumes WF: \( \text{wf } R \)
assumes INIT: \( I s_0 \)
\[
\begin{align*}
\text{assumes } & \text{STEP: } \forall s. \llbracket I \ s; \ \text{beal } b \ s \rrbracket \implies \wp \ \pi (c) \ (\lambda \ s'. \ I \ s' \wedge (s', s) \in R) \ s \\
\text{assumes } & \text{FINAL: } \forall s. \llbracket I \ s; \ \lnot \text{beal } b \ s \rrbracket \implies Q \ s \\
\text{shows } & \wp \ \pi (\text{WHILE } b \ DO \ c) \ Q \ s_0
\end{align*}
\]

\section{2.10 Modularity of Programs}
Adding more procedures does not change the semantics of the existing ones.

\textbf{lemma} map-leD: \( m \subseteq m' \implies m \ t = \text{Some v} \implies m' \ t = \text{Some v} \)

\textbf{lemma} big-step-mono-prog:
\begin{itemize}
  \item \text{assumes } \pi \subseteq m \ \pi'
  \item \text{assumes } \pi: (c, s) \Rightarrow t
  \item \text{shows } \pi': (c, s) \Rightarrow t
\end{itemize}

Wrapping a set of recursive procedures into a procedure scope

\textbf{lemma} localize-recursion:
\[ \pi': (\text{PScope } \pi \ c, s) \Rightarrow t \iff \pi: (c, s) \Rightarrow t \]

\section{2.11 Strongest Postcondition}
\textbf{context fixes } \pi :: \text{program}
\textbf{begin}
\textbf{definition} \( sp \ P \ c \ t \equiv \exists s. \ P \ s \wedge \pi: (c, s) \Rightarrow t \)

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

\textbf{Intuition: There exists an old value \( vx \) for the assigned variable}

\textbf{lemma} sp-arraycpy-eq: \( sp \ P \ (\text{\( x[]\)::=\( y \)}) \ t \iff (\exists vx. \ \text{let } s = t(\text{\( x:=vx \)}) \text{ in } t \ x = \text{\( s \ y \wedge P \ s \)}) \)

\textbf{lemma} sp-arraycpy-eq': \( sp \ P \ (\text{\( x[]\)::=\( y \)}) \ t \iff t \ x = t \ y \wedge (\exists vx. \ P (t(\text{\( x:=vx \),} y:=t \ x))) \)

\textbf{Version with renaming of assigned variable}

\textbf{lemma} sp-arraycpy-eq': \( sp \ P \ (\text{\( x[]\)::=\( y \)}) \ t \iff t \ x = t \ y \wedge (\exists vx. \ P (t(\text{\( x:=vx \),} y:=t \ x))) \)

\textbf{lemma} sp-skip-eq: \( sp \ P \ \text{SKIP} \ t \iff P \ t \)

\textbf{lemma} sp-seq-eq: \( sp \ P \ (c_1; c_2) \ t \iff sp \ (sp \ P \ c_1) \ c_2 \ t \)

\textbf{end}
\textbf{end}

\pagebreak
2.12 Hoare-Triples

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

**Definition** $HT$

where $HT \pi P c Q \equiv (\forall s_0. P s_0 \rightarrow \mathit{wp} \pi c (Q s_0) s_0)$

**Definition** $HT$-partial

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

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

**Lemma** $HT$-conseq:

- assumes $HT \pi P c Q$
- assumes $\land_{s.} P^\prime s \rightarrow P s$
- assumes $\land_{s_0} s. [P s_0; P^\prime s_0; Q s_0 s] \rightarrow Q^\prime s_0 s$
- shows $HT \pi P^\prime c Q^\prime$

**Proof**

**Lemma** $HT$-partial-conseq:

- assumes $HT$-partial $\pi P c Q$
- assumes $\land_{s.} P^\prime s \rightarrow P s$
- assumes $\land_{s_0} s. [P s_0; P^\prime s_0; Q s_0 s] \rightarrow Q^\prime s_0 s$
- shows $HT$-partial $\pi P^\prime c Q^\prime$

**Proof**

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

**Lemma** $\mathit{wp}$-modularity-rule:

\[ [HT \pi P c Q; P s; (\land_{s.} Q s s' \rightarrow Q' s')] \Rightarrow \mathit{wp} \pi c Q' s \]

**Proof**

2.12.1 Sets of Hoare-Triples

**Type-synonym** $htset = ((\text{state} \rightarrow \text{bool}) \times \text{com} \times (\text{state} \rightarrow \text{state} \rightarrow \text{bool}))$ 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** $HT \pi P body Q$
- **Shows** $HT \pi (wp \pi prologue P) (\prologue {}; body) (\lambda s_0. \ wp \pi prologue (\lambda s_0. \ Q s_0 \ s) s_0)$

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

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

**Lemma adjust-epilogue:**

- **Assumes** $HT \pi P body Q$
- **Assumes** $TERMINATES: \forall s. \ \exists t. \ \pi: (epilogue, s) \Rightarrow t$
- **Shows** $HT \pi P (body {}; epilogue) (\lambda s_0. \ sp (Q s_0) epilogue)$

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

**Lemma adjust-scope:**

- **Assumes** $HT \pi P body Q$
- **Shows** $HT \pi (\lambda s. \ P <<< | s) (SCOPE body) (\lambda s_0. \ \exists l. \ Q (<< | s_0)) (<|l|s>)$

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** $RL: \ \forall P c Q s_0. \ \\{ HTset-r (\lambda c' s'. ((c', s'), (c, s_0)) \in R ) \pi \Theta; (P, c, Q) \in \Theta; \ P s_0 \} \implies wp \pi c (Q s_0) s_0$

**Lemma HT-simple-recursiveI:**

- **Assumes** $wf R$
- **Assumes** $\forall s. \ [HT \pi (\lambda s'. (f s', f s) \in R \land P s') \ c Q; \ P s] \implies wp \pi c (Q s) s$
- **Shows** $HT \pi P c Q$
lemma HT-simple-recursive-procI:
assumes wf R
assumes \( \forall s. [\text{HT} \pi (\lambda s'. (f s', f s) \in R \land P s') (\text{PCall} p) \ Q; P s] \)
\implies wp \pi (\text{PCall} p) (Q s) s
shows HT \pi P (\text{PCall} p) Q
\langle proof \rangle

lemma
assumes wf R
assumes \( \forall s. P p Q. [\\]
\( \\\\forall P', p', Q'. (P', p', Q') \in \Theta \)
\( \implies \text{HT} \pi (\lambda s'. ((p', s'), (p, s)) \in R \land P' s') (P, p, Q) \in \Theta; P s \)
\( ] \implies wp \pi (\text{PCall} p) (Q s) s \)
shows \( \forall (P, p, Q) \in \Theta. \text{HT} \pi P (\text{PCall} p) Q \)
\langle proof \rangle

2.13 Completeness of While-Rule

Idea: Use wlp as invariant

lemma wlp-whileI-complete:
assumes wlp \pi (WHILE b DO c) \ Q s_0
obtains I where
\( I \ s_0 \)
\( \\forall s. I \ s \implies \text{if bval} b \ s \text{ then wlp } \pi c \ I \ s \text{ else } Q \ s \)
\langle proof \rangle

Idea: Remaining loop iterations as variant

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

lemma count-it-determ:
count-it \( \pi b c s n \implies \text{count-it } \pi b c s n' \implies n' = n \)
\langle proof \rangle

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

definition variant \( \pi b c s \equiv \text{THE } n. \text{count-it } \pi b c s n \)
lemma variant-decreases:
assumes STEPb: bval b s
assumes STEPC: \( \pi: (c, s) \Rightarrow s' \)
assumes TERM: \( \pi: (WHILE b \ DO c, s') \Rightarrow t \)
shows variant \( \pi \ b \ c \ s' \prec \) variant \( \pi \ b \ c \ s \)
⟨proof⟩

lemma wp-whileI'-complete:
fixes \( \pi \ b \ c \)
defines \( R \equiv \text{measure} \ (\text{variant} \ \pi \ b \ c) \)
assumes wp \( \pi \) \( (WHILE \ b \ DO \ c) \ Q \ s_0 \)
obtains \( I \) where
\[ wp \ R \]
\[ I \ s_0 \]
\[ \forall s \ . \ I \ s \implies \text{if} \ \text{bval} \ b \ s \ \text{then} \ wp \ \pi \ c \ (\lambda \ s'. \ I \ s' \land (s', s) \in R) \ s \ \text{else} \ Q \ s \]
⟨proof⟩

end

3 Annotated Syntax

theory Annotated-Syntax
imports Semantics
begin

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

named-theorems vcg-annotation-defs ⟨Definitions of Annotations⟩

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

definition ANNOTATION \equiv \lambda x. \ x

3.1 Annotations

The specification parser must interpret the annotations in the program.

definition WHILE-annotI :: \((\text{state} \Rightarrow \text{bool}) \Rightarrow \text{bexp} \Rightarrow \text{com} \Rightarrow \text{com}\)

\[ ((\text{WHILE} \ {-} \ - \ DO \ {-}) \ [0, 0, 61] \ 61) \]
where \[\text{vcg-annotation-defs}: \text{WHILE-annotI (I::state} \Rightarrow \text{bool}) \equiv \text{While}\]
lemmas annotate-whileI = WHILE-annotI-def [symmetric]

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

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

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

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

3.2 Hoare-Triples for Annotated Commands

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

type-synonym HT'-type = program ⇒ (state ⇒ bool) ⇒ (state ⇒ com) ⇒ (state ⇒ state⇒bool) ⇒ bool

definition HT'-partial :: HT'-type
  where HT'-partial π P c Q ≡ (∀ s₀. P s₀ −→ wlp π (c s₀) (Q s₀) s₀)

definition HT' :: HT'-type
  where HT' π P c Q ≡ (∀ s₀. P s₀ −→ wp π (c s₀) (Q s₀) s₀)

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

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

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

type-synonym 'a Θelem-t = (state⇒'a) × (state⇒bool) × (state⇒com) × (state⇒state⇒bool)

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

definition HT'set-r :: program ⇒ 'a Θelem-t set ⇒ bool where
HT'set-r r π Θ ≡ (∃ (n, (P, c, Q)) ∈ Θ. HT' (λs. r n s ∧ P s) c Q)
lemma \( HT'\text{setI} \):
\[
\begin{align*}
&\text{assumes } \langle w R \rangle \\
&\text{assumes } RL: \forall f \ P \ c \ Q \ s_0. \ [ HT'\text{set-r} (\lambda f' \ s'. \ ((f' \ s'), (f \ s_0)) \in R )] \\
&\pi: (f, (P, c, Q)) \in \Theta; \ P \ s_0 \quad \Longrightarrow \quad wp \pi (cs_0) (Qs_0) s_0 \\
&\text{shows } HT'\text{set} \pi \Theta
\end{align*}
\]
\( \langle \text{proof} \rangle \)

lemma \( HT'\text{setD} \):
\[
\begin{align*}
&\text{assumes } HT'\text{set} \pi (\text{insert} (f, (P, c, Q)) \Theta) \\
&\text{shows } HT' \pi P c Q \text{ and } HT'\text{set} \pi \Theta
\end{align*}
\]
\( \langle \text{proof} \rangle \)

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.

\[
\text{procedure-spec } \text{div-ab} \ (a, b) \ \text{returns } c
\]
\[
\begin{align*}
&\text{assumes } (b \neq 0) \\
&\text{ensures } (c = a_0 \ \text{div} \ b_0) \\
&\text{defines } (c = a/b)
\end{align*}
\]
\( \langle \text{proof} \rangle \)

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 \text{returns} declaration is optional, by default, nothing is returned. Multiple values can be returned by \text{returns} \((x_1, \ldots, x_n)\).

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

\textbf{Postcondition} An Isabelle formula over the return variables, and parameter names suffixed with \(0\).
Program Text  The procedure body, in a C-like syntax.

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

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

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

thm div-ab-spec — Final theorem proved

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

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

The precondition is a function state ⇒ bool. It starts with a series of variable bindings that map program variables to logical variables, followed by precondition that was specified, wrapped in a BB-PROTECT constant, which serves as a tag for the VCG, and is defined as the identity (BB-PROTECT ≡ λa. a).

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

procedure-spec use-div-ab(a) returns r assumes ⟨a≠0⟩ ensures ⟨r=1⟩ defines ⟨r = div-ab(a,a): ⟨proof⟩⟩

4.1.1 Variant and Invariant Annotations

Loops must be annotated with variants and invariants.

procedure-spec mult-ab(a,b) returns c assumes ⟨True⟩ ensures c=a0*b0 defines ⟨

if ⟨a<0⟩ { a = −a; b = −b};
c=0;
while ⟨a>0⟩
@variant ⟨a⟩
@invariant ⟨θ≤a ∧ a≤|a0| ∧ c = ( |a0| − a) * b0 * sgn a0⟩
\[ c = c + b; \]
\[ a = a - 1 \]
\}
\)
\langle proof \rangle

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

The variant must be an expression of type \( int \), which decreases with every loop iteration and is always \( \geq 0 \).

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

### 4.1.2 Recursive Procedures

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

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

Recursive invocations inside the specification have to be tagged by the \texttt{rec} keyword.

```
recursive-spec
odd-imp(n) returns b assumes n \geq 0 ensures \( b 
eq 0 \iff \text{odd n}_0 \)
variant \( \langle n \rangle \)
defines if \( (n==0) \) \( b=0 \) else \( b=\text{rec even-imp(n-1)} \)
and
even-imp(n) returns b assumes n \geq 0 ensures \( b \neq 0 \iff \text{even n}_0 \)
variant \( \langle n \rangle \)
defines if \( (n==0) \) \( b=1 \) else \( b=\text{rec odd-imp(n-1)} \);
\langle proof \rangle
```

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

```
thm odd-imp-spec even-imp-spec
```

### 4.2 The VCG

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

**procedure-spec** `mult-ab(a, b)` **returns** `c` **assumes** ⟨True⟩ **ensures**
\[
c = a_0 \times b_0
\]
**defines**
\[
\text{if } (a < 0) \{ a = -a; b = -b \};
\]
\[
c = 0;
\]
\[
\text{while } (a > 0)
\]
\[
\text{variant } (a)
\]
\[
\text{invariant } (0 \leq a \leq |a_0|) \wedge c = (|a_0| - a) \times b_0 \times \text{sgn } a_0
\]
\[
\{ c = c + b; a = a - 1 \}
\]
\langle proof \rangle

### 4.3 Advanced Features

#### 4.3.1 Custom Termination Relations

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

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

**procedure-spec** `mult-ab''(a, b)` **returns** `c` **assumes** ⟨True⟩ **ensures**
\[
c = a_0 \times b_0
\]
**defines**
\[
\text{if } (a < 0) \{ a = -a; b = -b \};
\]
\[
c = 0;
\]
\[
\text{while } (a > 0)
\]
\[
\text{relation } (\text{measure } \text{nat})
\]
\[
\text{variant } (a)
\]
\[
\text{invariant } (0 \leq a \leq |a_0|) \wedge c = (|a_0| - a) \times b_0 \times \text{sgn } a_0
\]
\[
\{ c = c + b; a = a - 1 \}
\]
\langle proof \rangle

**recursive-spec relation** ⟨measure nat⟩

`odd-imp'(n)` **returns** `b` **assumes** `n \geq 0` **ensures** ⟨`b \neq 0 ↔ odd n_0`⟩

**variant** ⟨n⟩

**defines**
\[
\text{if } (n = 0) \{ b = \theta \} \text{ else } b = \text{rec even-imp'}(n-1);
\]
and
even-imp′(n) returns b assumes n≥0 ensures \(b \neq 0 \leftrightarrow \text{even}\)

\(n_0\) variant (n)
defines if \((n=0)\) b=1 else \(b=\text{rec even-imp′}(n-1)\)
(proof)

4.3.2 Partial Correctness

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

procedure-spec (partial) nonterminating() returns a
assumes True
ensures \(a=0\)
defines while \((a \neq 0)\) @invariant (True)
a=a−1
(proof)

4.3.3 Arrays

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

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

lemma array-sum-aux: \(l_0 \leq l \Rightarrow \{l_0..<l+1\} = \text{insert} \ l \ \{l_0..<l\}\)
for \(l_0 \ l :: \text{int}\) (proof)

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

4.4 Proving Techniques

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

4.4.1 Auxiliary Lemmas

Prove auxiliary lemmas, and try to keep the actual proof of the specification small. As a rule of thumb: All VCs that cannot

---

\(^1\)Adding partial correctness for recursion is possible, however, compared to total correctness, showing that the prove rule is sound requires some effort that we have not (yet) invested.
be solved by a simple auto invocation should go to an auxiliary lemma.

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

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

### 4.4.2 Inlining

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

### 4.4.3 Functional Refinement

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

### 4.4.4 Data Refinement

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

### 4.5 Troubleshooting

We list a few common problems and their solutions here

#### 4.5.1 Invalid Variables in Annotations

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

Note the highlighting of unused variables in the following example

```isar
procedure-spec foo(x1,x2) returns y assumes x1>x2+x3 ensures y = x1+x2 defines 
  y=0;
  while (x1>0)
    @variant (y + x3)
    @invariant (y>x3)
    { x1=x2
      }
  }
⟨proof⟩
```

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

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

### 4.5.2 Wrong Annotations

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

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

```isar
procedure-spec (partial) foo () assumes True ensures True defines
  while (n>0) @variant ⟨n⟩ @invariant ⟨True⟩
  { n=n–1 }
⟨proof⟩
```

### 4.5.3 Calls to Undefined Procedures

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

This is an (incomplete) list of missing features.

4.6.1 Elaborate Warnings and Errors

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

4.6.2 Static Type Checking

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

4.6.3 Structure Types

Every variable is an integer arrays. Plain integer variables are implemented as macros on top of this, by referring to index 0. The most urgent addition to increase usability would be record types. With them, we could model encapsulation and data refinement more explicitly, by collecting all parts of a data structure in a single (record-typed) variable.

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

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

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 expres-
sions, 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) refine-
ment, and hinting the VCG towards the abstract algorithm struc-
ture.
We believe that we can add ghost variables with annotations on
top of the VCG, without actually changing the program seman-
tics.

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 sep-
eration logic, he might actually add pointers and memory allo-
cation to IMP2 in the near future.

end

5 Introduction to IMP2-VCG, based on IMP

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

This document briefly introduces the extensions of IMP2 over
IMP.

5.1 Fancy Syntax

Standard Syntax
**definition** \( \text{exp-count-up1} \equiv \)

\[
\begin{align*}
''a'' & := N 1 ; ; \\
''c'' & := N 0 ; ; \\
\text{WHILE} \text{ Cmpop} (\text{<}) (V ''c'') (V ''n'') \text{ DO } ( \\
''a'' & := \text{Binop} (+) (N 2) (V ''a'') ; ; \\
''c'' & := \text{Binop} (+) (V ''c'') (N 1))
\end{align*}
\]

Fancy Syntax

**definition** \( \text{exp-count-up2} \equiv \text{imp} \)

— Initialization

\( a = 1 ; ; \)
\( c = 0 ; ; \)

while \( (c<n) \) { — Iterate until \( c \) has reached \( n \)

\( a=2*a ; — \text{Double } a \)
\( c=c+1 — \text{Increment } c \)
}

**lemma** \( \text{exp-count-up1} = \text{exp-count-up2} \)

\( \langle \text{proof} \rangle \)

### 5.2 Operators and Arrays

We reflect arbitrary Isabelle functions into the syntax:

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

**thm** \( \text{aval}.\text{simps} \text{bval}.\text{simps} \)

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

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

New commands:

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

**term** \( \langle ''a''[i] := v \rangle — \text{Standard syntax} \)
**term** \( \langle \text{imp} \ a[i] = v \rangle — \text{Fancy syntax} \)

**lemma** \( \langle \text{Assign} ''x'' v = \text{AssignIdx} ''x'' (N 0) v \rangle \langle \text{proof} \rangle \)
**term** \( \langle ''x'' := v \rangle \text{ term } \langle \text{imp} \ x = v+1 \rangle \)

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.
term ⟨ArrayCpy "d" "s"⟩ — Copy whole array. Both operands are variable names.

term ⟨"d"[] ::= "s"[] term imp(d = s)⟩

term ⟨ArrayClear "a"⟩ — Initialize array to all zeroes.

term ⟨CLEAR "a"[] term imp(clear a[])⟩

Semantics of these is straightforward


5.3 Local and Global Variables

term ⟨is-global⟩ term ⟨is-local⟩ — Partitions variable names

term ⟨<s₁|s₂>⟩ — State with locals from s₁ and globals from s₂

term ⟨SCOPE c⟩ term ⟨imp(scope { skip })⟩ — Execute c with fresh set of local variables

thm big-step.Scope

5.3.1 Parameter Passing

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

context fixes f :: com begin
term ⟨imp((r₁,r₂) = f(x₁,x₂,x₃))⟩
end

5.4 Recursive procedures

term ⟨PCall "name"⟩

thm big-step.PCall

5.4.1 Procedure Scope

Execute command with local set of procedures

term ⟨PScope π c⟩

thm big-step.PScope

5.4.2 Syntactic sugar for procedure call with parameters

term ⟨imp((r₁,r₂) = rec name(x₁,x₂,x₃))⟩

5.5 More Readable VCs

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

35
lemma \(s_0 \ "n" \ 0 \geq 0 \implies \text{wlp} \ \pi \ \text{exp-count-up1} (\lambda s \ "a" \ 0 = 2 \cdot \text{nat} (s_0 \ "n" \ 0)) \ s_0\)

\begin{proof}\end{proof}

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

\begin{proof}\end{proof}

5.6 Specification Commands

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

Old way of proving a specification:

\begin{align*}
\text{lemma } & \text{let } n = s_0 \ "n" \ 0 \ \text{in } n \geq 0 \\
& \implies \text{wlp} \ \pi \ \text{exp-count-up1} (\lambda s \ "a" \ 0 = 2 \cdot \text{nat} (n_0)) \ s_0 \\
\end{align*}

\begin{proof}\end{proof}

\begin{align*}
\text{lemma } & \text{VAR} (s \ x) \ P = (\text{let } v = s \ x \ \text{in } P \ v) \ \langle \text{proof} \rangle \\
\end{align*}

IMP2 specification commands

\begin{align*}
\text{program-spec } & (\text{partial}) \ \text{exp-count-up} \\
& \text{assumes } 0 \leq n \quad \text{— Precondition. Use variable names of program.} \\
& \text{ensures } a = 2 \cdot \text{nat} n_0 \quad \text{— Postcondition. Use variable names of programs. Suffix with } \cdot 0 \text{ to refer to initial state} \\
& \text{defines } \\
& \quad a = 1; \\
& \quad c = 0; \\
& \quad \text{while } (c < n) \\
& \quad \text{\@invariant } (n = n_0 \land a = 2 \cdot \text{nat} c \land 0 \leq c \land c \leq n) \quad \text{— Invar annotation. Variable names and suffix } \cdot 0 \text{ for variables from initial state.} \\
& \quad \{ \\
& \quad \quad a = 2 \cdot a; \\
& \quad \quad c = c + 1 \\
& \quad \} \\
& \quad \langle \text{proof} \rangle \\
\end{align*}

\begin{proof}\end{proof}

\text{thm } \text{exp-count-up-spec} \\
\text{thm } \text{exp-count-up-def} \\

\begin{proof}\end{proof}

\text{procedure-spec } \text{exp-count-up-proc}(n) \ \text{returns } a \\

36
assumes $0 \leq n$
ensures $a = 2 \cdot \text{nat } n_0$
defines

\begin{verbatim}
  a = 1;
c = 0;
while (c < n)
  @invariant $\langle n = n_0 \land a = 2 \cdot \text{nat } c \land 0 \leq c \land c \leq n \rangle$
  @variant $\langle n - c \rangle$
  \{ 
    a = 2 \times a;
    c = c + 1
  \}
\end{verbatim}

\langle proof \rangle

Simple Recursion

\texttt{recursive-spec}

\texttt{exp-rec}(n) returns $a$ assumes $0 \leq n$ ensures $a = 2 \cdot \text{nat } n_0$ variant $n$
defines \texttt{if} ($n == 0$) $a = 1$ else \texttt{\{t = rec exp-rec(n - 1); a = 2 * t\}}\texttt{\}}\texttt{\}}

\langle proof \rangle

Mutual Recursion: See Examples

end

\texttt{theory} Examples
\texttt{imports ../IMP2 ../lib/IMP2-Aux-Lemmas}
\texttt{begin}

6 Examples

lemmas \texttt{nat-distros} = \texttt{nat-add-distros} \texttt{nat-diff-distros} \texttt{Suc-diff-le} \texttt{nat-mult-distros} \texttt{nat-div-distros}

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

\texttt{program-spec} \texttt{exp-count-up}
assumes $0 \leq n$
ensures \( a = 2^{\text{nat}\, n_0} \)
defines (\(\langle a = 1; c = 0; \text{while (c < n)} @\text{variant} \langle n-c \rangle @\text{invariant} \langle 0 \leq c \land c \leq n \land a = 2^{\text{nat}\, c} \rangle \{ G-par = a; \text{scope} \{ a = G-par; a = 2 \ast a; G-ret = a \}; a = G-ret; c = c + 1 \} \rangle \langle \text{proof} \rangle \))

\text{program-spec \( \text{sum-prog} \)}
\text{assumes} \( n \geq 0 \) \text{ensures} \( s = \sum \{ 0..n_0 \} \)
defines (\(\langle s = 0; i = 0; \text{while (i < n)} @\text{variant} \langle n_0 - i \rangle @\text{invariant} \langle n_0 = n \land 0 \leq i \land i \leq n \land s = \sum \{ 0..i \} \rangle \{ i = i + 1; s = s + i \} \rangle \langle \text{proof} \rangle \))

\text{program-spec \( \text{sq-prog} \)}
\text{assumes} \( n \geq 0 \) \text{ensures} \( a = n_0 \ast n_0 \)
defines (\(\langle a = 0; z = 1; i = 0; \text{while (i < n)} @\text{variant} \langle n_0 - i \rangle @\text{invariant} \langle n_0 = n \land 0 \leq i \land i \leq n \land a = i \ast i \land z = 2 \ast i + 1 \rangle \{ a = a + z; z = z + 2; i = i + 1 \} \rangle \langle \text{proof} \rangle \))

\text{fun factorial :: int} \Rightarrow \text{int where}\)
\( \text{factorial} \, i = (\text{if} \, i \leq 0 \, \text{then} \, 1 \, \text{else} \, i \ast \text{factorial} \, (i - 1)) \)
program-spec factorial-prog
assumes \( n \geq 0 \) ensures \( a = \text{factorial} \ n \)
defines \( a = 1; \)
\( i = 1; \)
while \( (i \leq n) \)
\( \langle \text{variant} \ (n_0 + 1 - i) \rangle \)
\( \langle \text{variant} \ (n_0 = n \land 1 \leq i \land i \leq n + 1 \land a = \text{factorial} \ (i - 1)) \rangle \)
\{ \)
\( a = a \ast i; \)
\( i = i + 1 \)
\} \langle proof \rangle

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

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

lemmas [simp del] = fib.simps

With precondition
program-spec fib-prog
assumes \( n \geq 0 \) ensures \( a = \text{fib} \ n \)
defines \( a = 0; \)
\( b = 1; \)
\( i = 0; \)
while \( (i < n) \)
\( \langle \text{variant} \ (n_0 - i) \rangle \)
\( \langle \text{variant} \ (n = n_0 \land 0 \leq i \land i \leq n \land a = \text{fib} \ i \land b = \text{fib} \ (i + 1)) \rangle \)
\{ \)
\( c = b; \)
\( b = a + b; \)
\( a = c; \)
\( i = i + 1 \)
\} \langle proof \rangle

Without precondition, returning 0 for negative numbers
program-spec \texttt{fib-prog}'
\textbf{assumes} True \textbf{ensures} a = \text{fib} \ n_0
\textbf{defines} \begin{align*}
a & = 0; b = 1; \\
i & = 0; \\
\text{while} \ (i < n) \quad & \text{variant} \ (n_0 - i) \\
& \text{invariant} \ (n = n_0 \land (0 \leq i \land i \leq n \lor n_0 < 0 \land i = 0) \land a = \text{fib} \ i \land b = \text{fib} \ (i + 1)) \end{align*}
\begin{align*}
\{ & c = b; \\
& b = a + b; \\
& a = c; \\
& i = i + 1 \\
\} \quad \langle \text{proof} \rangle
\end{align*}

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} \texttt{exp-count-down}
\textbf{assumes} $0 \leq n$
\textbf{ensures} $a = 2^\text{nat} \ n_0$
\textbf{defines} \begin{align*}
a & = 1; \\
\text{while} \ (n > 0) \quad & \text{variant} \ (n) \\
& \text{invariant} \ (\text{let} \ c = n_0 - n \text{ in } 0 \leq n \land n \leq n_0 \land a = 2^\text{nat} \ c) \\
\{ & a = 2 \ast a; \\
& n = n - 1 \\
\} \quad \langle \text{proof} \rangle
\end{align*}

6.1.3 Approximate from Below

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

The invariant states that the $r-1$ is not too big. When the loop terminates, $r-1$ is not too big, but $r$ is already too big, so $r-1$ is the desired value (rounding down).
The variant measures the gap that we have to the correct result. Note that the loop will do a final iteration, when the result has been reached exactly. We account for that by adding one, such that the measure also decreases in this case.

**program-spec** `sqr-approx-below`  
assumes $\theta \leq n$  
ensures $\theta \leq r \land r^2 \leq n_0 \land n_0 < (r+1)^2$  
defines $\langle r = 1; \land (r \geq n) \rangle$  
\[\langle \text{variant} \langle n + 1 - r \cdot r \rangle \rangle\]  
\[\langle \text{invariant} \langle \theta \leq r \land (r-1)^2 \leq n_0 \rangle \rangle\]  
\[\langle \text{variant} \langle r = r + 1 \rangle \rangle\]  
\[\langle \text{invariant} \langle r = r - 1 \rangle \rangle\]  
\[\langle \text{proof} \rangle\]  

### 6.1.4 Bisection

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

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

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

**lemma** $\forall h \cdot l \cdot n_0 :: \text{int}.$  
\[\langle \text{invar-final}\rangle; \theta \leq n_0; \land 1 + l < h; \theta \leq l; l < h; l \times l \leq n_0; n_0 < h \times h \rangle\]  
\[\implies n_0 < l^2 + (l \times l + l \times 2)\]  
\[\langle \text{proof} \rangle\]  

**program-spec** `sqr-bisect`  
assumes $\theta \leq n$  
ensures $r^2 \leq n_0 \land n_0 < (r+1)^2$  
defines $\langle l = 0; h = n+1 \rangle$  
\[\langle \text{variant} \langle h - l \rangle \rangle\]  
\[\langle \text{invariant} \langle 0 \leq l \land l < h \land l^2 \leq n \land n < h^2 \rangle \rangle\]  
\[\{ \{ m = (l + h) / 2; \land \text{if } (m \times m \leq n) l = m \text{ else } h = m \}\}; \quad r = l \]$
6.2 Debugging

6.2.1 Testing Programs

Stepwise

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

Or all steps at once

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

6.3 More Numeric Algorithms

6.3.1 Euclid’s Algorithm (with subtraction)

thm gcd.commute gcd-diff1

program-spec euclid1
assumes a>0 ∧ b>0
ensures a = gcd a0 b0
defines :
  while (a≠b)
    @invariant (gcd a b = gcd a0 b0 ∧ (a>0 ∧ b>0))
    @variant (a+b)
    { if (a<b) b = b−a
      else a = a−b
    }
⟩
⟨proof⟩

6.3.2 Euclid’s Algorithm (with mod)

thm gcd-red-int[ symmetric]

program-spec euclid2
assumes a>0 ∧ b>0
ensures a = gcd a0 b0
defines :
  while (b≠0)
    @invariant (gcd a b = gcd a0 b0 ∧ b≥0 ∧ a>0)
    @variant (b)
    { t = a;
      a = b;
  }
⟨proof⟩
\begin{equation}
    b = t \mod b
\end{equation}

\section{Extended Euclid’s Algorithm}

locale extended-euclid-aux-lemmas begin

lemma aux2:
    fixes a b :: int
    assumes \( b = t \cdot b_0 + s \cdot a_0 \)
    q = a \div b \gcd a b = \gcd a_0 b_0
    shows \( \gcd b \left( a - (a_0 \cdot (s \cdot q) + b_0 \cdot (t \cdot q)) \right) = \gcd a_0 b_0 \)

⟨proof⟩

lemma aux3:
    fixes a b :: int
    assumes \( b = t \cdot b_0 + s \cdot a_0 \)
    q > 0
    shows \( t \cdot (b_0 \cdot q) + s \cdot (a_0 \cdot q) \leq a \)

⟨proof⟩

end

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

program-spec euclid-extended
    assumes \( a > 0 \land b > 0 \)
    ensures \( \text{old-r} = \gcd a_0 b_0 \land \gcd a_0 b_0 = a_0 \cdot \text{old-s} + b_0 \cdot \text{old-t} \)
    defines : \( s = 0; \quad \text{old-s} = 1; \)
        \( t = 1; \quad \text{old-t} = 0; \)
        \( r = b; \quad \text{old-r} = a; \)
    while \( (r \neq 0) \)
        @invariant : \( \gcd \text{old-r} r = \gcd a_0 b_0 \land r \geq 0 \land \text{old-r} > 0 \)
            \land \ a_0 \cdot \text{old-s} + b_0 \cdot \text{old-t} = \text{old-r} \land a_0 \cdot s + b_0 \cdot t = r \)
        @variant \( r \)
        \{ \)
            quotient = old-r / r;
            temp = old-r;
            old-r = r;
            r = temp - quotient \* r;
            temp = old-s;
            old-s = s;
            s = temp - quotient \* s;
            temp = old-t;
        \}

43
\[ \text{old-t = t;} \]
\[ t = \text{temp - quotient} \times t \]
\}

⟨proof⟩

Non-Wikipedia version

context extended-euclid-aux-lemmas begin
lemma aux:
fixes a b x y:: int
assumes a = old-y \times b_0 + old-x \times a_0 \quad b = y \times b_0 + x \times a_0 \quad q = a \div b
shows
\[ a \mod b + (a_0 \times (x \times q) + b_0 \times (y \times q)) = a \]
⟨proof⟩
end

program-spec euclid-extended'
assumes a>0 ∧ b>0
ensures a = gcd a_0 \times b_0 ∧ gcd a_0 \times b_0 = a_0 \times x + b_0 \times y
defines
\{ x = 0; \]
y = 1;
old-x = 1;
old-y = 0;
while (b\neq 0)
@invariant ⟨
gcd a b = gcd a_0 \times b_0 ∧ b\geq 0 ∧ a>0 ∧ a = a_0 \times old-x + b_0 \times old-y ∧ b = a_0 \times x + b_0 \times y
⟩
@variant ⟨b⟩
\{ q = a \div b; \]
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
⟩
⟨proof⟩
6.3.4 Exponentiation by Squaring

**Lemma ex-by-sq-aux:**

- **Fixes** $x :: \text{int}$ and $n :: \text{nat}$
- **Assumes** $n \mod 2 = 1$
- **Shows** $x \cdot (x \cdot x) \cdot (n \div 2) = x^n$

**Proof**

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

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

**Program-Spec ex-by-sq**

- **Assumes** $n \geq 0$
- **Ensures** $r = x_0 \cdot \text{nat} n_0$
- **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 ($n \mod 2 = 1$)
        - $r = r \cdot x$
      - $x = x \cdot x$
      - $n = n / 2$
  - @**Proof**

6.3.5 Power-Tower of 2s

**Fun tower2 where**

- $\text{tower2} 0 = 1$
- $\text{tower2} (\text{Suc} n) = 2 \cdot \text{tower2} n$

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

**Program-Spec tower2-imp**

- **Assumes** ($m > 0$)
- **Ensures** ($a = \text{tower2'} m_0$)
- **Defines**:
  - $a = 1$
  - while ($m > 0$)
    - @**Variant** ($m$)
      - @**Invariant** ($0 \leq m \land m \leq m_0 \land a = \text{tower2'} (m_0 - m)$)
\{ 
  n = a; 
  
  a = 1; 
  while (n > 0) 
    \@variant \langle n \rangle 
    \@invariant \langle True \rangle — This will get ugly, there is no \( n_0 \) that we could use! 
    \{ 
      a = 2 * a; 
      n = n - 1 
    \}; 
  
  m = m - 1 
\} 
\langle proof \rangle

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

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

\section*{6.4 Array Algorithms}

\subsection*{6.4.1 Summation}

\textbf{program-spec} \textit{array-sum} \\
\textbf{assumes} \( l \leq h \) \\
\textbf{ensures} \( r = (\sum_{i = l_0 \ldots < h_0} a_0 \ i) \) \\
\textbf{defines} \langle 
  r = 0; 
  while \ (l < h) 
    \@invariant \langle l_0 \leq l \land l \leq h \land r = (\sum_{i = l_0 \ldots < l} a_0 \ i) \rangle 
    \@variant \langle h - l \rangle 
    \{ 
      r = r + a[l]; 
    \} 
\rangle 

46
6.4.2 Finding Least Index of Element

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

6.4.3 Check for Sortedness

term ran-sorted

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

if \( (l = h) \) \( r = 1 \) else \( r = 0 \))
⟩ ⟨proof⟩

6.4.4 Find Equilibrium Index

definition is-equil \( a \cdot l \cdot h \cdot i \equiv l \leq i \land i < h \land (\sum_{j = l}^{i} a \cdot j) = (\sum_{j = i}^{h} a \cdot j) \)

program-spec equilibrium
assumes \( l \leq h \)
ensures \( (\exists i. \text{is-equil } a \cdot l \cdot h \cdot i) \)
defines (usum \( = 0 \); i \( = l \);
while (i<h)
  @variant (h-i)
  @invariant (l≤i ∧ i≤h ∧ usum = (∑ j=l..<i. a j))
  {
    usum = usum + a[i]; i=i+1
  }
  i=l; lsum=0;
while (usum ≠ lsum ∧ i<h)
  @variant (h-i)
  @invariant (l≤i ∧ i≤h
    ∧ lsum=(∑ j=l..<i. a j)
    ∧ usum=(∑ j=i..<h. a j)
    ∧ (∀ j<i. ¬is-equal a l h j)
  )
  {
    lsum = lsum + a[i];
    usum = usum - a[i];
    i=i+1
  }
⟨proof⟩

6.4.5 Rotate Right

program-spec rotate-right
assumes 0<n
ensures ∀ i∈{0..<n}. a i = a₀ ((i-1) mod n)
defines i
  i = 0;
prev = a[n - 1];
while (i < n)
  @invariant
    0 ≤ i ∧ i ≤ n
    ∧ (∀ j∈{0..<i}. a j = a₀ ((j-1) mod n))
    ∧ (∀ j∈{i..<n}. a j = a₀ j)
    ∧ prev = a₀ ((i-1) mod n)
  )
  @variant ⟨n – i⟩
  {
    temp = a[i];
    a[i] = prev;
    prev = temp;
    i = i + 1
  }
⟨proof⟩
6.4.6 Binary Search, Leftmost Element

We first specify the pre- and postcondition

**Definition** \( \text{bin-search-pre } a \ l \ h \leftrightarrow l \leq h \land \text{ran-sorted } a \ l \ h \)  

**Definition** \( \text{bin-search-post } a \ l \ h \ x \ i \leftrightarrow l \leq i \land i \leq h \land (\forall i \in \{l..<i\}. \ a \ i < x) \land (\forall i \in \{i..<h\}. \ x \leq a \ i) \)  

Then we prove that the program is correct

**Program-Spec** \( \text{binsearch} \)

**Assumes** \( \langle \text{bin-search-pre } a \ l \ h \rangle \)

**Ensures** \( \langle \text{bin-search-post } a \ l_0 \ h_0 \ x_0 \ i_0 \rangle \)

**Defines**

\[
\{ \quad \begin{align*}
  m &= (l + h) / 2; \\
  &\text{if } (a[m] < x) \ l = m + 1 \\
  &\text{else } h = m
\end{align*} \}
\]

\( \langle \text{proof} \rangle \)

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

**Lemma**

**Assumes** \( \langle \text{bin-search-pre } a \ l \ h \ \text{bin-search-post } a \ l \ h \ x \ i \rangle \)

**Shows** \( \langle \text{bin-search-decide-membership}: x \in a^{\prime}\{l..<h\} \leftrightarrow (i < h \land x = a \ i) \rangle \)

**and** \( \langle \text{bin-search-leftmost}: x \notin a^{\prime}\{l..<i\} \rangle \)

(\( \langle \text{proof} \rangle \))

6.4.7 Naive String Search

**Program-Spec** \( \text{match-string} \)

**Assumes** \( l_1 \leq h_1 \)

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

**Defines**

\[
\{ \quad \begin{align*}
  i &= 0; \\
  \text{while } (l_1 + i < h_1 \land a[l + i] == b[l_1 + i]) &\rightarrow \,(h_1 - l_1) \land \invariant \langle (h_1 - (l_1 + i)) \rangle \}
\end{align*} \}
\]

\( i = i + 1 \)
lemma lran-eq-iff': lran a l \ (l + (h - l)) = lran a' l h
\iff l \leq h
\begin{proof}
(\forall i. \ 0 \leq i \land i < h - l \implies a \ (l + i) = a' \ (l + i))
\end{proof}

program-spec match-string'
assumes \( l \leq h \)
ensures \( i = h1 - l \iff \text{lran} \ a \ l \ (l + (h1 - l)) = \text{lran} \ b \ l1 \ h1 \)
for \( i \ h1 \ l1 \ a \ h1 \)
defines \( \text{inline match-string} \)
\begin{proof}
\end{proof}

program-spec substring
assumes \( l \leq h \land l1 \leq h1 \)
ensures \( \text{match} = 1 \iff (\exists j \in \{l0..<h0\}. \text{lran} \ a \ j \ (j + (h1 - l1))) = \text{lran} \ b \ l1 \ h1 \)
for \( a \ b \)
defines \( \text{inline substring} \)
\begin{proof}
\end{proof}

program-spec substring'
assumes \( l \leq h \land l1 \leq h1 \)
ensures \( \text{match} = 1 \iff (\exists j \in \{l0..<h0\}. \text{lran} \ a \ j \ (j + (h1 - l1))) = \text{lran} \ b \ l1 \ h1 \)
for \( a \ b \)
defines \( \text{match} = 0; \)
if \( (l + (h1 - l1) \leq h) \) \{ 
\begin{itemize}
  \item \( h \equiv h - (h1 - l1) + 1; \)
  \item inline substring
\end{itemize}
\}\begin{proof}
\end{proof}
program-spec substring''
assumes $l \leq h \land l_1 \leq h_1$
ensures $\text{match} = 1 \iff (\exists \ j \in \{l_0, < h_0 = (h_1 - l_1)\}. \ lran \ a \ j (j + (h_1 - l_1)) = lran \ b \ l_1 \ h_1)$
for $a[] b[]$
defines \langle match = 0; \ if \ (l + (h_1 - l_1) \leq h) \{ \ while \ (l + (h_1 - l_1) < h \land match == 0) \ @\text{invariant}\ l_0 \leq l \land l \leq h - (h_1 - l_1) \land match \in \{0,1\} \land \ (if \ match = 1 \ then \ lran \ a \ l \ (l + (h_1 - l_1)) = lran \ b \ l_1 \ h_1 \land l < h - (h_1 - l_1)) \ else \ (\forall \ j \in \{l_0, < l\}. \ lran \ a \ j (j + (h_1 - l_1)) \neq lran \ b \ l_1 \ h_1)) \} \langle \text{proof} \rangle

\begin{align*}
\text{lemma \ lan-split:} & \quad lran \ a \ l \ h = lran \ a \ l \ p \ @ \ lan \ a \ p \ h \ \text{if} \ l \leq p \ p \leq h \\
\langle \text{proof} \rangle & \end{align*}

\begin{align*}
\text{lemma \ lan-eq-append-iff:} & \quad lran \ a \ l \ h = as \ @ \ bs \iff (\exists \ i. \ l \leq i \land i \leq h \land as = lran \ a \ l \ i \land bs = lran \ a \ i \ h) \ \text{if} \ l \leq h \\
\langle \text{proof} \rangle & \end{align*}

\begin{align*}
\text{lemma \ lan-split':} & \quad (\exists j \in \{l, h\ - (h_1 - l_1)\}. \ lran \ a \ j (j + (h_1 - l_1)) = lran \ b \ l_1 \ h_1) \\
= (\exists \ as \ bs. \ lan \ a \ l \ h = as \ @ \ lan \ b \ l_1 \ h_1 \ @ \ bs) \ \text{if} \ l \leq h \ l_1 \leq h_1 \\
\langle \text{proof} \rangle & \end{align*}

\begin{align*}
\text{program-spec substring-final} & \quad \text{assumes} \ l \leq h \land 0 \leq \text{len} \\
\text{ensures} \ match = 1 \iff (\exists \ as \ bs. \ lan \ a \ l_0 \ h_0 = as \ @ \ lan \ b \ 0 \ \text{len} \ @ \ bs) \\
\text{for} \ l \ h \ \text{len} \ \text{match} \ a[] b[] \\
\text{defines} \ d1 = 0; \ h1 = \text{len}; \ \text{inline substring'}; \\
\langle \text{proof} \rangle & \end{align*}
6.4.8 Insertion Sort

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

**definition** insort-insert-post \( l \ j \ a \_0 \ a \ i \)

\[ \iff \text{(let } key = a \_0 \ j \text{ in} \]
\[ \quad i \in \{ l-1..<j \} \quad \text{— } i \text{ is in range} \]
\[ \quad \land (\forall k \in \{ l..i \}, \ a \ k = a \_0 \ k) \]
\[ \quad \land a \ (i+1) = key \]
\[ \quad \land (\forall k \in \{ i+2..j \}, \ a \ k = a \_0 \ (k-1)) \]
\[ \quad \land a = a \_0 \ \text{on} \ -\{l..j\} \]

— Content of new array

\[ \land (i \geq l \rightarrow a \ i \leq key) \quad \text{— Element at } i \text{ smaller than } key, \text{ if it exists} \]
\[ \land (\forall k \in \{ i+2..j \}, \ a \ k > key) \]

— Placement of key

\[ \land (i \geq l \rightarrow a \ i \leq key) \quad \text{— Element at } i \text{ smaller than } key, \text{ if it exists} \]
\[ \land (\forall k \in \{ i+2..j \}, \ a \ k > key) \quad \text{— Elements } \geq i+2 \text{ greater than } key \]

\) for \( l \ j \ i :: \int \text{ and } a \_0 \ a :: \int \Rightarrow \int \)

**program-spec** insort-insert

**assumes** \( l < j \)

**ensures** insort-insert-post \( l \ j \ a \_0 \ a \ i \)

**defines**

\[
\text{key} = a[j]; \\
i = j - 1; \\
\text{while } (i \geq l \land a[i] > key)
\]

@**variant** \( i - l + 1 \)

@**invariant** \( l - 1 \leq i \land i < j \)

\[ \land (\forall k \in \{ l..i \}, \ a \ k = a \_0 \ k) \]
\[ \land (\forall k \in \{ i+2..j \}, \ a \ k > key \land a \ k = a \_0 \ (k-1)) \]
\[ \land a = a \_0 \ \text{on} \ -\{l..j\} \]

\[
\{ \\
\quad a[i+1] = a[i]; \\
\quad i = i - 1 \\
\}
\]

\[ a[i+1] = key \]

\( \land \langle \text{proof} \rangle \)

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

Invoking insort-insert will sort in the element

**lemma** insort-insert-sorted:

**assumes** \( l < j \)

52
Invoking `insort-insert` will only mutate the elements

**Lemma `insort-insert-ran1`:**

- Assumes `l < j` and `insort-insert-post l j a a' i`.
- Shows `ran-sorted a l j`.

**Proof**

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

**Lemma `insort-insert-ran2`:**

- Assumes `l < j < h` and `insort-insert-post l j a a' i`.
- Shows `mset-ran a' {l..<h} = mset-ran a {l..<h}` (is ?thesis2).

**Proof**

Finally, we specify and prove correct the outer loop.

**Program-Spec `insort`:**

- Assumes `l < h`.
- Ensures `ran-sorted a l h \land mset-ran a \ {l..<h} = mset-ran a_0 \ {l..<h}`.
- Defines (a)
  
  \[
  j = l + 1; \quad \text{while} \ (j < h)
  \]

  \[
  \@variant \ (h-j)
  \@invariant (\ l < j \land j \leq h \land \text{ran-sorted} \ a \ l j \land mset-ran \ a \ \{l..<h\} = mset-ran \ a_0 \ \{l..<h\}) \quad \text{Array is sorted up to } j \quad \text{Elements in range only permuted}
  \]

  \[
  \land \ a = a_0 \ \text{on} \ \{-l..<h\}
  \]

  \[
  \{\ 
  \text{inline } \text{insort-insert};
  
  j = j + 1
  \}
  \]

**Proof**

### 6.4.9 Quicksort

**Procedure-Spec `partition-aux(a,l,h,p)` returns `(a,i)`**
assumes $l \leq h$

ensures $\text{mset-ran } a_{0 \cdots h_0} = \text{mset-ran } a_{l \cdots h_0}$
\begin{align*}
&\land (\forall j \in \{l_0 \cdots i\}. a_j < p_0) \\
&\land (\forall j \in \{i \cdots 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 \cdots h_0\}}
\end{align*}

defines $\langle \quad \rangle$

$i = l; j = l;$

while $(j < h)$

@invariant $\langle l \leq i \land i \leq j \land j \leq h \\
\land \text{mset-ran } a_{0 \cdots h_0} = \text{mset-ran } a_{l \cdots h_0} \\
\land (\forall k \in \{l \cdots i\}. a_k < p) \\
\land (\forall k \in \{i \cdots j\}. a_k \geq p) \\
\land (\forall k \in \{j \cdots h_0\}. a_k = a_k) \\
\land a_0 = a_{\text{on} \{l_0 \cdots h_0\}} \rangle$

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

\{ $\langle \text{proof} \rangle$

\}

\langle \text{proof} \rangle$

procedure-spec \text{partition}(a,l,h,p) returns $(a,i)$

assumes $l < h$

ensures $\text{mset-ran } a_{0 \cdots h_0} = \text{mset-ran } a_{l \cdots h_0}$
\begin{align*}
&\land (\forall j \in \{l_0 \cdots i\}. a_j < a_i) \\
&\land (\forall j \in \{i \cdots 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 \cdots h_0\}}
\end{align*}

defines $\langle \quad \rangle$

$p = a[h - 1];$

$(a,i) = \text{partition-aux}(a,l,h-1,p);$ 

$a[h - 1] = a[i];$

$a[i] = p$

\langle \text{proof} \rangle$

lemma \text{quicksort-sorted-aux}:

assumes $\text{BOUNDS}: l \leq i < h$

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

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

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

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

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

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

⟨proof⟩

lemma quicksort-mset-aux:
assumes $B$: $l_0 \leq i < h_0$
assumes $R1$: $\text{mset-ran } a \{l_0..<i\} = \text{mset-ran } aa \{l_0..<i\}$
assumes $E1$: $a = aa \text{ on } \{l_0..<i\}$
assumes $R2$: $\text{mset-ran } aa \{i+1..<h_0\} = \text{mset-ran } ab \{i+1..<h_0\}$
assumes $E2$: $aa = ab \text{ on } \{i+1..<h_0\}$
shows $\text{mset-ran } a \{l_0..<h_0\} = \text{mset-ran } ab \{l_0..<h_0\}$

⟨proof⟩

recursive-spec quicksort$(a,l,h)$ returns $a$
assumes $\text{True}$
ensures $\text{ran-sorted } a \ l \ h \land \text{mset-ran } a_0 \{l_0..<h_0\} = \text{mset-ran } a \{l_0..<h_0\} \land a_0 = a \text{ on } \{l_0..<h_0\}$
variant $h-l$
defines $\{I\}$
if $(l<h)$ {
  $(a,i) = \text{partition}(a,l,h,a[l])$;
  $a = \text{rec quicksort}(a,l,i)$;
  $a = \text{rec quicksort}(a,i+1,h)$
}

⟨proof⟩

6.5 Data Refinement

6.5.1 Filtering

program-spc array-filter-negative
assumes $l \leq h$
ensures $\text{ran } a \ l \ i = \text{filter } (\lambda x. x \geq 0) (\text{ran } a_0 \ l \ h_0)$
defines $\{i=1; j=l; \text{while } (j<h) \} @\text{invariant}$

55
\begin{align*}
l \leq i \land i \leq j \land j \leq h \\
\land \ lran \ a \ l \ i = \ filter \ (\lambda x. \ x \geq 0) \ (lran \ a \_0 \ l \ j) \\
\land \ lran \ a \ j \ h = \ lran \ a \_0 \ j \ h
\end{align*}

\textit{@variant} \langle h-j \rangle

\{ \\
\text{if} \ (a[j] \geq 0) \ \{a[i] = a[j]; \ i \leftarrow i+1\}; \\
\quad j \leftarrow j+1 \\
\}

\langle \textit{proof} \rangle

\subsection*{6.5.2 Merge Two Sorted Lists}

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

\begin{verbatim}
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)
\end{verbatim}

\begin{enumerate}
    \item \textbf{lemma} \textit{merge-add-simp}: \textit{simp}: \textit{merge} \ x s i = \ x s \langle \textit{proof} \rangle
\end{enumerate}

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

\begin{enumerate}
    \item \textbf{lemma} \textit{merge-sorted}:
    \textit{assumes} \textit{sorted} \ x s \textit{sorted} \ y s \\
    \textit{shows} \textit{sorted} \ (\textit{merge} \ x s \ y s) \land \textit{set} \ (\textit{merge} \ x s \ y s) = \textit{set} \ x s \cup \textit{set} \ y s \\
    \langle \textit{proof} \rangle
\end{enumerate}

\begin{enumerate}
    \item \textbf{lemma} \textit{merge-mset}: \textit{mset} \ (\textit{merge} \ x s \ y s) = \textit{mset} \ x s + \textit{mset} \ y s \\
    \langle \textit{proof} \rangle
\end{enumerate}

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

\begin{enumerate}
    \item \textbf{lemma} \textit{merge-eq}: \textit{xs} \neq [] \lor \textit{ys} \neq [] \implies \textit{merge} \ x s \ y s = ( \\
    \text{if} \ \textit{ys} = [] \lor \ (\textit{xs} \neq [] \land \textit{hd} \ x s < \textit{hd} \ y s) \text{ then } \textit{hd} \ x s \ # \ \textit{merge} \ (\textit{tl} \ x s) \ y s \\
    \text{else } \textit{hd} \ y s \ # \ \textit{merge} \ x s (\textit{tl} \ y s) \\
    ) \\
    \langle \textit{proof} \rangle
\end{enumerate}

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 \textit{bran} function to map from the implementation level to the list level. Moreover, the invariant of the implementation, \( l \leq h \), is carried through explicitly.
program-spec merge-imp'
assumes \( l1 \leq h1 \land l2 \leq h2 \)
ensures let \( ms = \text{lran } m \ 0 \ j \); \( xs_0 = \text{lran } a1_0 \ l1_0 \ h1_0 \); \( ys_0 = \text{lran } a2_0 \ l2_0 \ h2_0 \) in
\( j \geq 0 \land ms = \text{merge } xs_0 \ ys_0 \)
defines (inline merge-imp)

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

⟨proof⟩

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

procedure-spec merge-imp \((a1,l1,h1,a2,l2,h2)\) returns \((m,j)\)
assumes \( l1 \leq h1 \land l2 \leq h2 \land \text{sorted } (\text{lran } a1 \ l1 \ h1) \land \text{sorted } (\text{lran } a2 \ l2 \ h2) \)
ensures let \( ms = \text{lran } m \ 0 \ j \) in
\( j \geq 0 \land \text{sorted } ms \land mset \ ms = mset (\text{lran } a1_0 \ l1_0 \ h1_0) + mset (\text{lran } a2_0 \ l2_0 \ h2_0) \)
for \( l1 \ h1 \ l2 \ h2 \ a1[] \ a2[] \ m[] \ j \)
defines (inline merge-imp)
thm merge-imp-spec

thm merge-imp-def

lemma [named-ss wog-bb]:
\[ \text{UNIV} \cup a = \text{UNIV} \]
\[ a \cup \text{UNIV} = \text{UNIV} \]
\langle proof \rangle

lemma merge-msets-aux: \[[l \leq m; m \leq h] \implies \text{mset} (lran a l m) + \text{mset} (lran a m h) = \text{mset} (lran a l h) \]
\langle proof \rangle

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 :
if \((l == h)\) \(j = 0\)
else if \((l + 1 == h)\) 
\(b[0] = a[l];\)
\(j = 1\)
else 
\(m = (h + l) / 2;\)
\((a1, h1) = \text{rec mergesort} (a, l, m);\)
\((a2, h2) = \text{rec mergesort} (a, m, h);\)
\((b, j) = \text{merge-imp} (a1, 0, h1, a2, 0, h2)\)
\}
\rangle
\langle proof \rangle

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

context notes [simp] = set-of-def begin
lemma set-of-empty[simp]: set-of (λ. 0) = {} ⟨proof⟩
lemma set-of-insert[simp]: x ≠ 0 ⟹ set-of (a(i:=x)) = insert i (set-of a) ⟨proof⟩
lemma set-of-remove[simp]: set-of (a(i:=0)) = set-of a – {i} ⟨proof⟩
lemma set-of-mem[simp]: i ∈ set-of a ←→ a i ≠ 0 ⟨proof⟩
end

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

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

procedure-spec bv-insert (x, b) returns b
  assumes True
  ensures ⟨set-of b = insert x_0 (set-of b_0)⟩
  defines ⟨b[x] = 1⟩
  ⟨proof⟩

procedure-spec bv-remove (x, b) returns b
  assumes True
  ensures ⟨set-of b = set-of b_0 – {x_0}⟩
  defines ⟨b[x] = 0⟩
  ⟨proof⟩

procedure-spec bv-elem (x,b) returns r
  assumes True
  ensures ⟨r ≠ 0 ←→ x_0 ∈ set-of b_0⟩
  defines ⟨r = b[x]⟩

59
procedure-spec dedup\' (a,l,h) returns (a,l,i)
assumes (l \leq h) ensures (set (tran a l i) = set (tran a_0 l_0 h_0)) \land
distinct (tran a l i)
for b[]
defines b = bv-init();

i=l; j=l;

while (j < h)
  @variant (h-j)
  @invariant (l \leq i \land i \leq j \land j \leq h)
  \land set (tran a l i) = set (tran a_0 l j)
  \land distinct (tran a l i)
  \land tran a j h = tran a_0 j h
  \land set-of b = set (tran a l i)
  
  { mem = bv-elem (a[j],b);
    if (mem == 0) {
      a[i] = a[j]; i=i+1; b = bv-insert(a[j],b)
    };
    j=j+1
  }

\langle proof \rangle

6.6 Recursion

6.6.1 Recursive Fibonacci

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

\langle proof \rangle
6.6.2 Homeier’s Cycling Termination

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

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

and
coast (n,m) returns () assumes ⟨n≥0 ∧ m≥0⟩ ensures True variant (n+m+1):
defines :
  G = G + n;
  if (n<m) rec coast (n,m−1) else rec pedal (n,m)

⟨proof⟩
```

6.6.3 Ackermann

```
fun ack :: nat ⇒ nat ⇒ nat where
  ack 0 n = n+1
  | ack m 0 = ack (m−1) 1
  | ack m n = ack (m−1) (ack m (n−1))

lemma ack-add-simps[simp]:
m≠0 ⇒ ack m 0 = ack (m−1) 1
[[m≠0; n≠0]] ⇒ ack m n = ack (m−1) (ack m (n−1))
⟨proof⟩
```

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

⟨proof⟩
```

61
6.6.4 McCarthy’s 91 Function

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

\[
\text{recursive-spec } p91(y) \text{ assumes } \text{True} \quad \text{ensures } \begin{cases} 100 < y_0 & \text{then } G = y_0 - 10 \\ \text{else } G = 91 \end{cases} \quad \text{variant } 101 - y
\]

for \( G \)

\[
\begin{cases}
\text{if } (100 < y) & G = y - 10 \\
\text{else } \{ & \\
& \text{rec } p91 \ (y+11); \\
& \text{rec } p91 \ (G) \\
\}
\end{cases}
\]

\langle \text{proof} \rangle

6.6.5 Odd/Even

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

\langle \text{proof} \rangle

\text{even-imp } (a) \text{ returns } b \\
\text{assumes } \text{True} \\
\text{ensures } b \neq 0 \iff \text{even } a_0 \\
\text{variant } |a| \\
\text{defines } ( \\
\quad \text{if } (a = 0) & b = 1 \\
\quad \text{else if } (a < 0) & b = \text{rec } \text{odd-imp } (a+1) \\
\quad \text{else } b = \text{rec } \text{odd-imp } (a-1) \\
\}
\langle \text{proof} \rangle

\text{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 \( y \).

\[
\text{recursive-spec relation } (\text{measure } \text{nat} <^\text{lex} > \text{less-than}) \\
\text{product } () \text{ assumes } \text{True} \quad \text{ensures } \langle GZ = GZ_0 + GX_0 \ast GY_0 \rangle \text{ variant } (|GY|,1::\text{nat})
\]
for $GX$ $GY$ $GZ$
defines
\{
  e = \text{even-imp}(GY);
  if (e\neq 0) \text{rec evenproduct()}\ else\ \text{rec oddproduct()}
\}\nand
\text{oddproduct()}\ \text{assumes}\ \langle\text{odd }GY\rangle\ \text{ensures}\ \langle GZ = GZ_0 + GX_0\times GY_0\rangle
\text{variant}\ \langle |GY|,0::\text{nat}\rangle

for $GX$ $GY$ $GZ$
defines
\{
  if (GY<0) \{
    GY = GY + 1;
    GZ = GZ - GX
  \} \ else \{
    GY = GY - 1;
    GZ = GZ + GX
  \};
  \text{rec evenproduct()}
\}\nand
\text{evenproduct()}\ \text{assumes}\ \langle\text{even }GY\rangle\ \text{ensures}\ \langle GZ = GZ_0 + GX_0\times GY_0\rangle
\text{variant}\ \langle |GY|,0::\text{nat}\rangle

for $GX$ $GY$ $GZ$
defines
\{
  if (GY\neq 0) \{
    GX = 2\times GX;
    GY = GY / 2;
    \text{rec product()}
  \}
\}\n\langle\text{proof}\rangle

6.7 Graph Algorithms
6.7.1 DFS

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

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

Indexes: $... \mid i \mid i+1 \mid ... \mid i+n \mid ...$

Data: $... \mid n \mid s1 \mid ... \mid sn \mid ...$

definition \text{succs} where
\text{succs }a\ i \equiv a \cdot \{i+1...<a \ i\} \ \text{for}\ a :: \text{int} \Rightarrow \text{int}
definition Edges where
Edges a ≡ {(i, j). j ∈ succs a i}

procedure-spec push′ (x, stack, ptr) returns (stack, ptr)
assumes ptr ≥ 0 ensures \( [x_0] \land ptr = ptr_0 + 1 \)
defines \( stack[ptr] = x; ptr = ptr + 1 \)
⟨proof⟩

procedure-spec push (x, stack, ptr) returns (stack, ptr)
assumes ptr ≥ 0 ensures \( stack' \{0..<ptr\} = \{x_0\} \cup stack_0' \{0..<ptr_0\} \land ptr = ptr_0 + 1 \)
for stack[]
defines \( stack[ptr] = x; ptr = ptr + 1 \)
⟨proof⟩

program-spec get-succs
assumes j ≤ stop ∧ stop = a (j - 1) ∧ 0 ≤ i
ensures \( stack' \{0..<i\} = \{x, (j_0 - 1, x) \in Edges \land x \not\in set\text{-}of \ visited\} \cup stack_0' \{0..<i_0\} \land i_0 ≥ i_0 \)
for i j stop stack[] a[] visited[]
defines
while (j < stop)
@invariant \( stack' \{0..<i\} = \{x, x \in a \land x \not\in set\text{-}of \ visited\} \cup stack_0' \{0..<i_0\} \land j ≤ stop \land i_0 ≤ i \land j_0 ≤ j \)
@variant \((stop - j)\)
{
  succ = a[j];
  is-elem = bv\text{-}elem(succ, visited);
  if (is\text{-}elem == 0) {
    (stack, i) = push (succ, stack, i)
 );
  j = j + 1
}
⟨proof⟩

procedure-spec pop (stack, ptr) returns (x, ptr)
assumes ptr ≥ 1 ensures \( stack_0' \{0..<ptr_0\} = stack_0' \{0..<ptr\} \cup \{x\} \land ptr_0 = ptr + 1 \)
for stack[]
defines \( ptr = ptr - 1; x = stack[ptr] \)
⟨proof⟩
procedure-spec stack-init () returns i
assumes True ensures (i = 0)
defines (i = 0)
<proof>

lemma Edges-empty:
Edges a "" \{i\} = {} if i + 1 \geq a i
<proof>

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

lemma reachability-invariant:
assumes reachable: (s, x) \in (Edges a)^* and closed: \forall v\in visited. Edges a "" \{v\} \subseteq visited and start: s \in visited
shows x \in visited
<proof>

program-spec (partial) dfs
assumes 0 \leq x \land 0 \leq s
ensures b = 1 \iff x \in (Edges a)^* "" \{s\} defines :
b = 0;
clear stack[];
i = stack-init();
(stack, i) = push (s, stack, i);
clear visited[];
while (b == 0 \land i \neq 0)
\@invariant: 0 \leq i \land (s \in stack \land \{0..<i\} \cup s \in set-of visited) \land (b = 0 \lor b = 1) \land (if b = 0 then stack \land \{0..<i\} \subseteq (Edges a)^* "" \{s\} \land (\forall v \in set-of visited. (Edges a)^* v \subseteq set-of visited \cup stack \land \{0..<i\})
\land (x \notin set-of visited)
else x \in (Edges a)^* "" \{s\})
)

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

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

lemma sub-insert-same-iff: s ⊆ insert x s ←→ x ∉ s

program-spec dfs1
assumes 0 ≤ x ∧ 0 ≤ s ∧ start = s ∧ edges = a
ensures b = 1 ←→ x ∈ (Edges a)* "{s}"
for visited[]
defines
b = 0;
— i will point to the next free space in the stack (i.e. it is the size of the stack)
i = 1;
— Initially, we put s on the stack.
stack[0] = s;
visited = bv-init();
while (b == 0 ∧ i ≠ 0)
©invariant (:
0 ≤ i ∧ (s ∈ stack ' {0..<i} ∨ s ∈ set-of visited) ∧ (b = 0 ∨ b = 1) ∧
set-of visited ⊆ (Edges edges)* "{start}" ∧ (if b = 0 then
stack ' {0..<i} ⊆ (Edges a)* "{s}"
∧ (∀ v ∈ set-of visited. (Edges a) * {v} ⊆ set-of visited ∪ stack ' {0..<i})
∧ (x ∉ set-of visited)
else x ∈ (Edges a)* "{s}\n)
©relation (finite-psupset ((Edges edges)* "{start}" <slex*> less-than)
©variant ((set-of visited, nat i)):
{ — Take the top most element from the stack.
  (next, i) = pop(stack, i);
  if (next == x) {
    — If it is the target, we are done.
    ...
\[ \text{visited} = \text{bv-insert}(\text{next}, \text{visited}); \]
\[
\text{b} = 1
\]
}\text{else}\{
\text{is-elem} = \text{bv-elem}(\text{next}, \text{visited});
\text{if} \ (\text{is-elem} \equiv 0) \{
\text{visited} = \text{bv-insert}(\text{next}, \text{visited});
\text{— Else, put its successors on the stack if they are not yet visited.}
\text{stop} = a[\text{next}];
\text{j} = \text{next} + 1;
\text{if} \ (j \leq \text{stop}) \{
\text{inline get-succs}
\}
\}
\}
\}
\}\langle \text{proof}\rangle
\]
\end

\end