Promela Formalization

By René Neumann

May 28, 2015

Abstract

We present an executable formalization of the language Promela, the description language for models of the model checker SPIN. This formalization is part of the work for a completely verified model checker (CAVA), but also serves as a useful (and executable!) description of the semantics of the language itself, something that is currently missing. The formalization uses three steps: It takes an abstract syntax tree generated from an SML parser, removes syntactic sugar and enriches it with type information. This further gets translated into a transition system, on which the semantic engine (read: successor function) operates.
# Contents

1 Introduction 3

2 Abstract Syntax Tree 6

3 Data structures as used in Promela 9
   3.1 Abstract Syntax Tree after preprocessing 9
   3.2 Preprocess the AST of the parser into our variant 12
   3.3 The transition system 25
   3.4 State 26
   3.5 Printing 27

4 Invariants for Promela data structures 30
   4.1 Bounds 31
   4.2 Variables and similar 31
   4.3 Invariants of a process 33
   4.4 Invariants of the global state 35
   4.5 Invariants of the program 36

5 Formalization of Promela semantics 37
   5.1 Misc Helpers 37
   5.2 Variable handling 38
   5.3 Expressions 41
   5.4 Variable declaration 45
   5.5 Folding 49
   5.6 Starting processes 51
   5.7 AST to edges 53
      5.7.1 Setup 57
   5.8 Semantic Engine 60
      5.8.1 Evaluation of Edges 60
      5.8.2 Executable edges 63
      5.8.3 Successor calculation 66
      5.8.4 Handle non-termination 71
   5.9 Finiteness of the state space 72
   5.10 Traces 72
      5.10.1 Printing of traces 73
   5.11 Code export 74

6 LTL integration 74
   6.1 LTL optimization 75
   6.2 Language of a Promela program 77
   6.3 Proposition types and conversion 79
1 Introduction

Promela [1] is a modeling language, mainly used in the model checker SPIN [2]. It offers a C-like syntax and allows to define processes to be run concurrently. Those processes can communicate via shared global variables or by message-passing via channels. Inside a process, constructs exist for non-deterministic choice, starting other processes and enforcing atomicity. It furthermore allows different means for specifying properties: LTL formulae, assertions in the code, never claims (i.e. an automata that explicitly specifies unwanted behavior) and others.

Some constructs found in Promela models, like `#include` and `#define`, are not part of the language Promela itself, but belong to the language of the C preprocessor. SPIN does not process those, but calls the C compiler internally to process them. We do not deal with them here, but also expect the sources to be preprocessed.

Observing the output of SPIN and examining the generated graphs often is the only way of determining the semantics of a certain construct. This is complicated further by SPIN unconditionally applying optimizations. For the current formalization we chose to copy the semantics of SPIN, including the aforementioned optimizations. For some constructs, we had to restrict the semantics, i.e. some models are accepted by SPIN, but not by this formalization. Those deviations are:

- `run` is a statement instead of an expression. SPIN here has a complicated set of restrictions unto where `run` can occur inside an expression. The sole use of it is to be able to get the ID of a spawned process. We omitted this feature to guarantee expressions to be free of side-effects.

- Variable declarations which got jumped over are seen as not existing. In SPIN, such constructs show surprising behavior:
  
  ```c
  int i; goto L; i = 5; L: printf("%d", i) yields 0, while goto L; int i = 5; L: printf("%d", i) yields 5.
  ```

  The latter is forbidden in our formalization (it will get rejected with “unknown variable i”), while the first behaves as in SPIN.

- Violating an `assert` does not abort, but instead sets the variable `__assert__` to true. This needs to be checked explicitly in the LTL formula. We plan on adding this check in an automatic manner.

- Types are bounded. Except for well-defined types like booleans, overflow is not allowed and will result in an error. The same holds for assigning a value that is outside the bounds. SPIN does not specify any explicit semantics here, but solely refers to the underlying C-compiler and its semantics. This might result in two models behaving differently on different systems when run with SPIN, while this formalization, due to the explicit bounds in the semantics, is not affected.
Additionally, some constructs are currently not supported, and the compilation will abort if they are encountered: \texttt{d_step}, \texttt{typedef}, remote references, bit-operations, \texttt{unsigned}, and property specifications except \texttt{ltl} and \texttt{assert}. Other constructs are accepted but ignored, because they do not change the behavior of a model: advanced variable scoping, \texttt{xr}, \texttt{xs}, \texttt{print*}, priorities, and visibility of variables.

Nonetheless, for models not using those unsupported constructs, we generate the very same number of states as SPIN does. An exception applies for large \texttt{goto} chains and when simultaneous termination of multiple processes is involved, as SPIN’s semantics is too vague here.

\texttt{theory Lexord-List}

\texttt{imports Main}

\texttt{begin}

\texttt{typedef }a \texttt{lexlist} = \{xs::}a \texttt{list. True}\}

\texttt{morphisms unlex Lex}

\texttt{(proof)}

\texttt{definition lexlist \equiv Lex}

\texttt{lemma lexlist-ext:}

\texttt{Lex xs = Lex ys \implies xs = ys}

\texttt{(proof)}

\texttt{lemma Lex-unlex [simp, code abstype]:}

\texttt{Lex (unlex lxs) = lxs}

\texttt{(proof)}

\texttt{lemma unlex-lexlist [simp, code abstract]:}

\texttt{unlex (lexlist xs) = xs}

\texttt{(proof)}

\texttt{definition list-less :: }a :: \texttt{ord list }\Rightarrow \texttt{'}a \texttt{list }\Rightarrow \texttt{bool where}

\texttt{list-less xs ys }\iff\texttt{ (xs, ys) }\in\texttt{ lexord }\{(u, v). u < v\}

\texttt{definition list-le where}

\texttt{list-le xs ys }\iff\texttt{ list-less xs ys }\lor\texttt{ xs = ys}

\texttt{lemma not-less-Nil [simp]: }\neg\texttt{ list-less x []}

\texttt{(proof)}

\texttt{lemma Nil-less-Cons [simp]: list-less [] (a }\#\texttt{ x)}

\texttt{(proof)}

\texttt{lemma Cons-less-Cons [simp]: list-less (a }\#\texttt{ x) (b }\#\texttt{ y) }\iff\texttt{ a < b }\lor\texttt{ a = b }\land\texttt{ list-less x y}

\footnote{This can be safely replaced by \texttt{atomic}, though larger models will be produced then.}
proof

lemma le-Nil [simp]: list-le x [] \iff x = []

lemma Nil-le-Cons [simp]: list-le [] x

lemma Cons-le-Cons [simp]: list-le (a # x) (b # y) \iff a < b \lor a = b \land list-le x y

lemma less-list-code [code]:
  list-less xs [] \iff False
  list-less [] (x # xs) \iff True
  list-less (x # xs) (y # ys) \iff x < y \lor x = y \land list-less xs ys

lemma less-eq-list-code [code]:
  list-le (x # xs) [] \iff False
  list-le [] xs \iff True
  list-le (x # xs) (y # ys) \iff x < y \lor x = y \land list-le xs ys

instantiation lexlist :: (ord) ord
begin

definition lexlist-less-def: xs < ys \iff list-less (unlex xs) (unlex ys)

definition lexlist-le-def: (xs :: - lexlist) \le ys \iff list-le (unlex xs) (unlex ys)

instance (proof)

lemmas lexlist-ord-defs = lexlist-le-def lexlist-less-def list-le-def list-less-def

end

instance lexlist :: (order) order
(proof)

instance lexlist :: (linorder) linorder
(proof)

end
2 Abstract Syntax Tree

theory PromelaAST
imports Main
begin

The abstract syntax tree is generated from the handwritten SML parser. This theory only mirrors the data structures from the SML level to make them available in Isabelle.

context
begin

⟨ML⟩

datatype binOp =
    BinOpAdd
  | BinOpSub
  | BinOpMul
  | BinOpDiv
  | BinOpMod
  | BinOpBitAnd
  | BinOpBitXor
  | BinOpBitOr
  | BinOpGr
  | BinOpLe
  | BinOpGEq
  | BinOpLEq
  | BinOpEq
  | BinOpNEq
  | BinOpShiftL
  | BinOpShiftR
  | BinOpAnd
  | BinOpOr

datatype unOp =
    UnOpComp
  | UnOpMinus
  | UnOpNeg

datatype expr =
    ExprBinOp binOp expr expr
  | ExprUnOp unOp expr
  | ExprCond expr expr expr
  | ExprLen varRef
  | ExprPoll varRef recvArg list
  | ExprRndPoll varRef recvArg list
  | ExprVarRef varRef
  | ExprConst integer

| ExprTimeOut  |
| ExprNP      |
| ExprEnabled expr |
| ExprPC expr |
| ExprRemoteRef String.literal |
| ExprGetPrio expr |
| ExprSetPrio expr expr |
| ExprFull varRef |
| ExprEmpty varRef |
| ExprNFull varRef |
| ExprNEmpty varRef |

\textbf{and} \ VarRef = VarRef String.literal
\hspace{1em} expr option
\hspace{1em} varRef option

\textbf{and} \ recvArg = RecvArgVar varRef
| RecvArgEval expr |
| RecvArgConst integer |

\textbf{datatype} range =
| RangeFromTo varRef |
| expr |
| RangeIn varRef varRef |

\textbf{datatype} varType =
| VarTypeBit |
| VarTypeBool |
| VarTypeByte |
| VarTypePid |
| VarTypeShort |
| VarTypeInt |
| VarTypeMType |
| VarTypeChan |
| VarTypeUnsigned |
| VarTypeCustom String.literal |

\textbf{datatype} varDecl =
| VarDeclNum String.literal |
| integer option |
| expr option |
| VarDeclChan String.literal |
| integer option |
| (integer * varType list) option |
| VarDeclUnsigned String.literal |
| integer |
expr option
| VarDeclMType String.literal
| integer option
| String.literal option

datatype decl =
Decl bool option
| varType
| varDecl list

datatype stmt =
| StmtIf (step) list
| StmtDo (step) list
| StmtFor range step list
| StmtAtomic step list
| StmtDStep step list
| StmtSelect range
| StmtSeq step list
| StmtSend varRef expr list
| StmtSortSend varRef expr list
| StmtRecv varRef recvArg list
| StmtRndRecv varRef recvArg list
| StmtReceX varRef recvArg list
| StmtSendX varRef recvArg list
| StmtAssign varRef expr
| StmtIncr varRef
| StmtDecr varRef
| StmtElse
| StmtBreak
| StmtGoTo String.literal
| StmtLabeled String.literal stmt
| StmtPrintF String.literal expr list
| StmtPrintM String.literal
| StmtRun String.literal expr list
| integer option
| StmtAssert expr
| StmtCond expr
| StmtCall String.literal varRef list

and step = StepStmt stmt stmt option
| StepDecl decl
| StepXR varRef list
| StepXS varRef list

datatype module =
| ProcType (integer option) option
3 Data structures as used in Promela

theory PromelaDatastructures
imports
  ../CAVA-Automata/CAVA-Base/CAVA-Base
  ../CAVA-Automata/CAVA-Base/Lexord-List
  PromelaAST
  ~/src/HOL/Library/IArray
  ../Deriving/Comparator-Generator/Compare-Instances
  ../CAVA-Automata/CAVA-Base/CAVA-Code-Target
begin

3.1 Abstract Syntax Tree after preprocessing

From the plain AST stemming from the parser, we’d like to have one containing more information while also removing duplicated constructs. This is achieved in the preprocessing step.

The additional information contains:

- variable type (including whether it represents a channel or not)
- global vs local variable
Also certain constructs are expanded (like for-loops) or different nodes in the AST are collapsed into one parametrized node (e.g. the different send-operations).

This preprocessing phase also tries to detect certain static errors and will bail out with an exception if such is encountered.

**datatype**

\[
\text{binOp} = \text{BinOpAdd} | \text{BinOpSub} | \text{BinOpMul} | \text{BinOpDiv} | \text{BinOpMod} | \text{BinOpGr} | \text{BinOpLe} | \text{BinOpGEq} | \text{BinOpLEq} | \text{BinOpEq} | \text{BinOpNEq} | \text{BinOpAnd} | \text{BinOpOr}
\]

**datatype**

\[
\text{unOp} = \text{UnOpMinus} | \text{UnOpNeg}
\]

**datatype**

\[
\text{expr} = \text{ExprBinOp} \text{ binOp} \text{ expr expr} | \text{ExprUnOp} \text{ unOp} \text{ expr} | \text{ExprCond} \text{ expr expr expr} | \text{ExprLen} \text{ chanRef} | \text{ExprVarRef} \text{ varRef} | \text{ExprConst} \text{ integer} | \text{ExprMConst} \text{ integer String.literal} | \text{ExprTimeOut} | \text{ExprFull chanRef} | \text{ExprEmpty chanRef} | \text{ExprPoll chanRef recvArg list bool}
\]

**and**

\[
\text{varRef} = \text{VarRef bool} \text{ String.literal expr option}
\]

**and**

\[
\text{chanRef} = \text{ChanRef varRef} — \text{explicit type for channels}
\]

**and**

\[
\text{recvArg} = \text{RecvArgVar varRef} | \text{RecvArgEval expr} | \text{RecvArgConst integer} | \text{RecvArgMConst integer String.literal}
\]

**datatype**

\[
\text{varType} = \text{VTBounded integer integer} | \text{VTCchan}
\]

Variable declarations at the beginning of a proctype or at global level.

**datatype**

\[
\text{varDecl} = \text{VarDeclNum integer integer}
\]
Variable declarations during a proctype.

data type procVarDecl = ProcVarDeclNum integer integer
      String.literal
      integer option
      expr option
    | ProcVarDeclChan String.literal
      integer option
      (integer * varType list) option


data type procArg = ProcArg varType String.literal


data type stmnt = StmtIf (step list) list
    | StmtDo (step list) list
    | StmtAtomic step list
    | StmtSeq step list
    | StmtSend chanRef expr list bool
    | StmtRecv chanRef recvArg list bool bool
    | StmtAssign varRef expr
    | StmtElse
    | StmtBreak
    | StmtSkip
    | StmtGoTo String.literal
    | StmtLabeled String.literal stmnt
    | StmtRun String.literal
      expr list
    | StmtCond expr
    | StmtAssert list

  and step = StepStmt stmnt stmnt option
           | StepDecl procVarDecl list
           | StepSkip


data type proc = ProcType (integer option) option
   String.literal
   procArg list
   varDecl list
   step list
 | Init varDecl list step list


type-synonym ltl = (*name*) String.literal × (*formula*) String.literal

type-synonym promela = varDecl list × proc list × ltl list
3.2 Preprocess the AST of the parser into our variant

We setup some functionality for printing warning or even errors. All those constants are logically `undefined`, but replaced by the parser for something meaningful.

```sml
consts
  warn :: String.literal ⇒ unit

abbreviation with-warn msg exp ≡ let - = warn (STR msg) in exp
abbreviation the-warn opt msg ≡ case opt of None ⇒ () | - ⇒ warn (STR msg)

usc: "Unsupported Construct"
definition [code del]: usc' c ≡ undefined
abbreviation usc c ≡ usc' (STR c)
definition [code del]: err' e = undefined
abbreviation err e ≡ err' (STR e)
abbreviation errv e v ≡ err' (STR e @@ v)
definition [simp, code del]: abort' msg f = f ()
abbreviation abort msg f ≡ abort' (STR msg) f
abbreviation abortv msg v f ≡ abort' (STR msg @@ v) f
```

code-printing
code-module PromelaUtils → (SML) ⟨⟨
structure PromelaUtils = struct
  exception UnsupportedConstruct of string
  exception StaticError of string
  exception RuntimeError of string
  fun warn msg = TextIO.print (Warning: "msg \\
  fun usc c = raise (UnsupportedConstruct c)
  fun err e = raise (StaticError e)
  fun abort msg - = raise (RuntimeError msg)
end ⟩⟩
| constant warn → (SML) PromelaUtils.warn
| constant usc' → (SML) PromelaUtils.usc
| constant err' → (SML) PromelaUtils.err
| constant abort' → (SML) PromelaUtils.abort
code-reserved SML PromelaUtils

⟨ML⟩

The preprocessing is done for each type on its own.

primrec ppBinOp :: AST.binOp ⇒ binOp
where
  ppBinOp AST.BinOpAdd = BinOpAdd
  ppBinOp AST.BinOpSub = BinOpSub
  ppBinOp AST.BinOpMul = BinOpMul
| ppBinOp AST.BinOpMod = BinOpMod
| ppBinOp AST.BinOpGr = BinOpGr
| ppBinOp AST.BinOpLe = BinOpLe
| ppBinOp AST.BinOpGEq = BinOpGEq
| ppBinOp AST.BinOpLEq = BinOpLEq
| ppBinOp AST.BinOpEq = BinOpEq
| ppBinOp AST.BinOpNEq = BinOpNEq
| ppBinOp AST.BinOpAnd = BinOpAnd
| ppBinOp AST.BinOpOr = BinOpOr
| ppBinOp AST.BinOpBitAnd = BinOpBitAnd
| ppBinOp AST.BinOpBitXor = BinOpBitXor
| ppBinOp AST.BinOpBitOr = BinOpBitOr
| ppBinOp AST.BinOpShiftL = BinOpShiftL

primrec ppUnOp :: AST.unOp ⇒ unOp
where
| ppUnOp AST.UnOpMinus = UnOpMinus
| ppUnOp AST.UnOpNeg = UnOpNeg
| ppUnOp AST.UnOpComp = UnOpComp

The data structure holding all information on variables we found so far.

type-synonym var-data =
  (String.literal, (integer option × bool)) lm (* channels *)
  × (String.literal, (integer option × bool)) lm (* variables *)
  × (String.literal, integer) lm (* mtypes *)
  × (String.literal, varRef) lm (* aliases (used for inlines *)

definition dealWithVar :: var-data ⇒ String.literal
  ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
  ⇒ (String.literal ⇒ integer option × bool ⇒ expr option ⇒ 'a)
  ⇒ (integer ⇒ 'a) ⇒ 'a
where
dealWithVar cvm n fC fV fM ≡ ()
  let (c,v,m,a) = cvm in
  let (n, idx) = case lm.lookup n a of
    None ⇒ (n, None)
    Some (VarRef - name idx) ⇒ (name, idx)
in
  case lm.lookup n m of
    Some i ⇒ (case idx of None ⇒ fM i
      | - ⇒ err' "Array subscript used on MType (via alias)."
    )
    | None ⇒ (case lm.lookup n v of
      Some g ⇒ fV n g idx
      | None ⇒ (case lm.lookup n c of
        Some g ⇒ fC n g idx
        | None ⇒ err' (String.implode ("Unknown variable referenced: " @

13
String.explode n))))

primrec enforceChan :: varRef + chanRef ⇒ chanRef where
  enforceChan (Inl _) = err "Channel expected. Got normal variable."
| enforceChan (Inr c) = c

fun liftChan :: varRef + chanRef ⇒ varRef where
  liftChan (Inl v) = v
| liftChan (Inr (ChanRef v)) = v

fun resolveIdx :: expr option ⇒ expr option ⇒ expr option where
  resolveIdx None None = None
| resolveIdx idx None = idx
| resolveIdx None aliasIdx = aliasIdx
| resolveIdx - - = err "Array subscript used twice (via alias)."

fun ppExpr :: var-data ⇒ AST.expr ⇒ expr and ppVarRef :: var-data ⇒ AST.varRef ⇒ varRef + chanRef and ppRecvArg :: var-data ⇒ AST.recvArg ⇒ recvArg where
  ppVarRef cvm (AST.VarRef name idx None) = dealWithVar cvm name ((λ name (-g) aIdx. let idx = map-option (ppExpr cvm) idx in
  Inr (ChanRef (VarRef g name (resolveIdx idx aIdx))))
| ppVarRef cvm (AST.VarRef - - (Some -)) = usc "next operation on variables"
| ppVarRef cvm (AST.ExprTimeOut) = ExprTimeOut
| ppVarRef cvm (AST.ExprConst c) = ExprConst c

| ppExpr cvm (AST.ExprBinOp bo l r) = ExprBinOp (ppBinOp bo) (ppExpr cvm l) (ppExpr cvm r)
| ppExpr cvm (AST.ExprUnOp uo e) = ExprUnOp (ppUnOp uo) (ppExpr cvm e)
| ppExpr cvm (AST.ExprCond c t f) = ExprCond (ppExpr cvm c) (ppExpr cvm t) (ppExpr cvm f)

| ppExpr cvm (AST.ExprLen v) = ExprLen (enforceChan (ppVarRef cvm v))
| ppExpr cvm (AST.ExprFull v) = ExprFull (enforceChan (ppVarRef cvm v))
| ppExpr cvm (AST.ExprEmpty v) = ExprEmpty (enforceChan (ppVarRef cvm v))

| ppExpr cvm (AST.ExprNFull v) = ExprUnOp UnOpNeg (ExprFull (enforceChan (ppVarRef cvm v)))
\[ \text{ppExp} \text{r cvm} (\text{AST.} \text{ExprNE}mpty \ v) = \]
\[ \text{ExprUnOp UnOpNeg (ExprEmpty (enforceChan (ppVarRef cvm v)))} \]

\[ \text{ppExp} \text{r cvm} (\text{AST.} \text{ExprVarRef} \ v) = (\]
\[ \text{let to-exp} = \lambda - . \text{ExprVarRef (liftChan (ppVarRef cvm v)) in} \]
\[ \text{case v of} \]
\[ \text{AST.} \text{VarRef name None None} \Rightarrow \]
\[ \text{dealWithVar cvm name} \]
\[ (\lambda - - . \text{to-exp()}) \]
\[ (\lambda - - . \text{to-exp()}) \]
\[ (\lambda. \text{ExprMConst i name}) \]
\[ | - \Rightarrow \text{to-exp()}) \]

\[ \text{ppExp} \text{r cvm} (\text{AST.} \text{ExprPoll} \ v \ es) = \]
\[ \text{ExprPoll (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm es) \text{False})} \]

\[ \text{ppExp} \text{r cvm} (\text{AST.} \text{ExprR}nd\text{Poll} \ v \ es) = \]
\[ \text{ExprPoll (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm es) \text{True})} \]

\[ \text{ppExp} \text{r cvm (AST.} \text{ExprNP} = \text{usc}'' \text{ExprNP}'' \]
\[ \text{ppExp} \text{r cvm (AST.} \text{ExprEnabled -} = \text{usc}'' \text{ExprEnabled}'' \]
\[ \text{ppExp} \text{r cvm (AST.} \text{ExprPC -} = \text{usc}'' \text{ExprPC}'' \]
\[ \text{ppExp} \text{r cvm (AST.} \text{ExprRemoteRef - -} = \text{usc}'' \text{ExprRemoteRef}'' \]
\[ \text{ppExp} \text{r cvm (AST.} \text{ExprGetPrio -} = \text{usc}'' \text{ExprGetPrio}'' \]
\[ \text{ppExp} \text{r cvm (AST.} \text{ExprSetPrio - -} = \text{usc}'' \text{ExprSetPrio}'' \]

\[ \text{ppRecvArg cvm (AST.} \text{RecvArgVar} \ v = (\]
\[ \text{let to-ra} = \lambda - . \text{RecvArgVar (liftChan (ppVarRef cvm v)) in} \]
\[ \text{case v of} \]
\[ \text{AST.} \text{VarRef name None None} \Rightarrow \]
\[ \text{dealWithVar cvm name} \]
\[ (\lambda - - . \text{to-ra()}) \]
\[ (\lambda - - . \text{to-ra()}) \]
\[ (\lambda. \text{RecvArgMConst i name}) \]
\[ | - \Rightarrow \text{to-ra()}) \]

\[ \text{ppRecvArg cvm (AST.} \text{RecvArgEval} \ e = \text{RecvArgEval (ppExpr cvm e)} \]
\[ \text{ppRecvArg cvm (AST.} \text{RecvArgConst} \ c = \text{RecvArgConst c) \}

\text{primrec ppVarT} \text{ype :: AST.} \text{varType} \Rightarrow \text{varType where} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeBit} = \text{VTBounded 0 1} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeBoolean = V} \text{TBounded 0 1} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeByte} = \text{VTBounded 0 255} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeP} \text{id} = \text{VTBounded 0 255} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeShort} = \text{VTBounded } -(2^{15}) (2^{15}) - 1 \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeInt} = \text{VTBounded } -(2^{31}) (2^{31}) - 1 \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeM} \text{type} = \text{VTBounded 1 255} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeChan} = \text{VTChan} \]
\[ \text{ppVarT} \text{ype AST.} \text{VarTypeUnsigned = usc}'' \text{VarTypeUnsigned}'' \]
\[ \text{ppVarT} \text{ype (AST.} \text{VarTypeCustom -} = \text{usc}'' \text{VarTypeCustom}'' \]

15
fun ppVarDecl :: var-data ⇒ varType ⇒ bool ⇒ AST.varDecl ⇒ var-data × varDecl

where

ppVarDecl (c,v,m,a) (VTBounded l h) g
(AST.VarDeclNum name sze init) = (case lm.lookup name v of
  Some - ⇒ errv "Duplicate variable " name
  | - ⇒ (case lm.lookup name a of
      Some - ⇒ errv "Variable name clashes with alias: " name
      | - ⇒ ((c, lm.update name (sze,g ) v, m, a),
             VarDeclNum l h name sze
             (map-option (ppExpr (c,v,m,a)) init)))
  | ppVarDecl - - g (AST.VarDeclNum name sze init) = errv "Assigning num to non-num"

| ppVarDecl (c,v,m,a) VTChan g
(AST.VarDeclChan name sze cap) = (let cap' = map-option (apsnd (map ppVarType)) cap in
  case lm.lookup name c of
    Some - ⇒ errv "Duplicate variable " name
    | - ⇒ (case lm.lookup name a of
        Some - ⇒ errv "Variable name clashes with alias: " name
        | - ⇒ ((lm.update name (sze, g ) c, v, m, a),
               VarDeclChan name sze cap')))
  | ppVarDecl - - g (AST.VarDeclChan name sze init) = errv "Assigning chan to non-chan"

| ppVarDecl (c,v,m,a) (VTBounded l h) g
(AST.VarDeclMType name sze init) = (let init = map-option (λmty.
  case lm.lookup mty m of
    None ⇒ errv "Unknown MType " mty
    | Some mval ⇒ ExprMConst mval mty) init in
  case lm.lookup name c of
    Some - ⇒ errv "Duplicate variable " name
    | - ⇒ (case lm.lookup name a of Some - ⇒ errv "Variable name clashes with alias: " name
    | - ⇒ ((c, lm.update name (sze,g ) v, m, a),
             VarDeclNum l h name sze init)))
  | ppVarDecl - - g (AST.VarDeclMType name sze init) = errv "Assigning num to non-num"

| ppVarDecl - - (AST.VarDeclUnsigned - - ) = usc "VarDeclUnsigned"

definition ppProcVarDecl
Some preprocessing functions enrich the \texttt{var-data} argument and hence return
a new updated one. When chaining multiple calls to such functions after another, we need to make sure, the var-data is passed accordingly. cvm-fold does exactly that for such a function g and a list of nodes ss.

**definition cvm-fold where**
\[
cvm-fold g cvm ss = \text{foldl} \ (\lambda(cvm,ss) s. \text{apsnd} \ (\lambda s'. ss@[(s')]) \ (g \ cvm s)) \ (\text{cvm, []}) ss
\]

**lemma cvm-fold-cong[fundef-cong]:**

**assumes** \(cvm = cvm'\)

**and** \(\text{stepss} = \text{stepss}'\)

**and** \(\forall x d. \ x \in \text{set stepss} \implies g d x = g' d x\)

**shows** \(cvm-fold g cvm \text{ stepss} = cvm-fold g' cvm' \text{ stepss}'\)

⟨proof⟩

**fun liftDecl where**

\[
liftDecl f g \text{ cvm} \text{ (AST.Decl vis t decls)} = (\text{let - = the-warn vis "Visibility in declarations not supported. Ignored." in let t = ppVarType t in cvm-fold (\lambda c. \text{f cvm t g} c) \text{ cvm} \text{ decls}})
\]

**definition ppDecl :: bool \Rightarrow var-data \Rightarrow AST.decl \Rightarrow var-data \times varDecl list**

**where**

\(\text{ppDecl} = \text{liftDecl ppVarDecl}\)

**definition ppDeclProc :: var-data \Rightarrow AST.decl \Rightarrow var-data \times procVarDecl list**

**where**

\(\text{ppDeclProc} = \text{liftDecl ppProcVarDecl False}\)

**definition ppDeclProcArg :: var-data \Rightarrow AST.decl \Rightarrow var-data \times procArg list**

**where**

\(\text{ppDeclProcArg} = \text{liftDecl ppProcArg False}\)

**definition incr :: varRef \Rightarrow stmnt where**

\(\text{incr v} = \text{StmtAssign v (ExprBinOp BinOpAdd (ExprVarRef v) (ExprConst 1))}\)

**definition decr :: varRef \Rightarrow stmnt where**

\(\text{decr v} = \text{StmtAssign v (ExprBinOp BinOpSub (ExprVarRef v) (ExprConst 1))}\)

Transforms for \((i : lb .. ub) \text{ steps into}\)

\{
  \text{i = lb; do}
  :: i <= ub -> steps; i++
:: else -> break
od

**definition** forFromTo :: varRef ⇒ expr ⇒ expr ⇒ step list ⇒ stmt where
forFromTo i lb ub steps = (�n
let
(* i = lb *)
loop-pre = StepStmt (StmtAssign i lb) None;
(* i ≤ ub *)
loop-cond = StepStmt (StmtCond
  (ExprBinOp BinOpLEq (ExprVarRef i) ub))
  None;
(* i++ *)
loop-incr = StepStmt (incr i) None;
(* i ≤ ub -> ...; i++ *)
loop-body = loop-cond # steps @ [loop-incr];
(* else -> break *)
loop-abort = [StepStmt StmtElse None, StepStmt StmtBreak None];
(* do :: i ≤ ub -> ...; else -> break od *)
loop = StepStmt (StmtDo [loop-body, loop-abort]) None
in
StmtSeq [loop-pre, loop])

Transforms (where a is an array with N entries) for (i in a) steps into

{ 
i = 0;
do 
:: i < N -> steps; i++
:: else -> break
od
}

**definition** forInArray :: varRef ⇒ integer ⇒ step list ⇒ stmt where
forInArray i N steps = (∨
let
(* i = 0 *)
loop-pre = StepStmt (StmtAssign i (ExprConst 0)) None;
(* i < N *)
loop-cond = StepStmt (StmtCond
  (ExprBinOp BinOpLe (ExprVarRef i)
   (ExprConst N)))
  None;
(* i++ *)
loop-incr = StepStmt (incr i) None;
(* i < N -> ...; i++ *)
loop-body = loop-cond # steps @ [loop-incr];
Transforms (where \( c \) is a channel) for \( \text{msg in } c \) steps into

\[
\begin{align*}
\{ & \text{byte } \text{:tmp: = 0;} \\
& \text{do} \\
& \quad :: \text{:tmp: < len(c) -> c?msg; c!msg; steps; \text{:tmp:++}} \\
& \quad :: \text{else -> break} \\
& \text{od} \\
\}
\]

\text{definition forInChan :: varRef } \Rightarrow \text{ chanRef } \Rightarrow \text{ step list } \Rightarrow \text{ stmt where}

\text{forInChan } \text{msg } c \text{ steps } = ( \text{let}

\begin{align*}
& (* \text{ byte :tmp: = 0 } *) \\
& \text{tmpStr = STR "':tmp:'";} \\
& \text{loop-pre = StepDecl} \\
& \quad [\text{ProcVarDeclNam 0 255 tmpStr None (Some (ExprConst 0))}]; \\
& \text{tmp = VarRef False tmpStr None;} \\
& (* \text{:tmp: < len(c) *}) \\
& \text{loop-cond = StepStmtnt (StmtntCond} \\
& \quad (\text{ExprBinOp BinOpLe (ExprVarRef tmp)} \\
& \quad \quad \quad \text{(ExprLen c)}) \\
& \quad \text{None;}) \\
& (* \text{:tmp:++ *}) \\
& \text{loop-incr = StepStmtnt (incr tmp) None;}
& (* \text{ c?msg } *) \\
& \text{recv = StepStmtnt (StmtntRecv c [ReceiveArgVar msg] False True) None;} \\
& (* \text{ c!msg } *) \\
& \text{send = StepStmtnt (StmtntSend c [ExprVarRef msg] False) None;} \\
& (* :tmp: < len(c) -> c?msg; c!msg; ... :tmp:++ *) \\
& \text{loop-body = [loop-cond, recv, send] @ steps @ [loop-incr];}
& (* \text{else -> break *}) \\
& \text{loop-abort = [StepStmtnt StmntElse None, StepStmtnt StmntBreak None];}
& (* \text{do :: :tmp: < len(c) -> ... :: else -> break od *}) \\
& \text{loop = StepStmtnt (StmtntDo [loop-body, loop-abort]) None}
\text{ in StmntSeq [loop-pre, loop]}
\end{align*}

\text{Transforms select (i : lb .. ub) into}

20
{  
i = lb;
do:: i < ub -> i++
:: break
od}
definition select :: varRef => expr => expr => stmt where
select i lb ub = (  
let
   (* i = lb *)
pre = StepStmt (StmtAssign i lb) None;
   (* i < ub *)
cond = StepStmt (StmtCond (ExprBinOp BinOpLe (ExprVarRef i) ub))
      None;
   (* i++ *)
incr = StepStmt (incr i) None;
   (* i < ub |-> i++ *)
loop-body = [cond, incr];
   (* break *)
loop-abort = [StepStmt StmtBreak None];
   (* do :: i < ub |-> ... :: break od *)
loop = StepStmt (StmtDo [loop-body, loop-abort]) None
in
   StmtSeq [pre, loop])
type-synonym inlines =
   (String.literal, String.literal list \times (var-data \Rightarrow var-data \times step list)) lm
type-synonym stmt-data =
   bool \times varDecl list \times inlines \times var-data
fun ppStep :: stmt-data => AST.step => stmt-data \times step
and ppStmt :: stmt-data => AST.stmt => stmt-data \times stmt
where
ppStep cvm (AST.StepStmt s u) = (  
let (cvm', s') = ppStmt cvm s in
   case u of None \Rightarrow (cvm', StepStmt s' None)
    | Some u \Rightarrow let (cvm'',u') = ppStmt cvm' u in
       (cvm'', StepStmt s' (Some u')))
| ppStep (False, ps, i, cvm) (AST.StepDecl d) =
    map-prod (\cvm. (False, ps, i, cvm)) StepDecl (ppDeclProc cvm d)
| ppStep (True, ps, i, cvm) (AST.StepDecl d) = (  
let (cvm', ps') = ppDecl False cvm d
in ((True, ps@ps', i, cvm'), StepSkip))
| ppStep (\cvm) (AST.StepXR -) =
   with-warn "StepXR not supported. Ignored."
      ((False, cvm), StepSkip)
| ppStep (\cvm) (AST.StepXS -) =
   with-warn "StepXS not supported. Ignored."
      ((False, cvm), StepSkip)
AST

(StmntBreak)

ppStmt (\(-,\_\text{com}\))

((False,\_\text{com}), StmntBreak)

ppStmt (\(-,\_\text{com}\))

((False,\_\text{com}), StmntElse)

ppStmt (\(-,\_\text{com}\))

((False,\_\text{com}), StmntGoTo l)

ppStmt (\(-,\_\text{com}\))

((False,\_\text{com}), StmntLabeled l s)

= apsnd (StmntLabeled l) (ppStmtt (False,\_\text{com}) s)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntCond e)

= ((False,ps,i,\_\text{com}), StmntCond (ppExpr cvm e))

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntAssert e)

= ((False,ps,i,\_\text{com}), StmntAssert (ppExpr cvm e))

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntAssign v e)

= ((False,ps,i,\_\text{com}), StmntAssign (liftChan (ppVarRef cvm v)) (ppExpr cvm e))

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntSend v es)

= ((False,ps,i,\_\text{com}), StmntSend (enforceChan (ppVarRef cvm v)) (map (ppExpr cvm) es) False)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntSortSend v es)

= ((False,ps,i,\_\text{com}), StmntSend (enforceChan (ppVarRef cvm v)) (map (ppExpr cvm) es) True)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntRecv v rs)

= ((False,ps,i,\_\text{com}), StmntRecv (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) rs) False True)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntRecvX v rs)

= ((False,ps,i,\_\text{com}), StmntRecv (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) rs) False False)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntRndRecv v rs)

= ((False,ps,i,\_\text{com}), StmntRndRecv (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) rs) True True)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntRndRecvX v rs)

= ((False,ps,i,\_\text{com}), StmntRndRecv (enforceChan (ppVarRef cvm v)) (map (ppRecvArg cvm) rs) True False)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntRun n p)

= (let - = the-warn p "Priorities for 'run' not supported. Ignored." in ((False,ps,i,\_\text{com}), StmntRun n (map (ppExpr cvm) es)))]

ppStmt (\(-,\_\text{com}\))

(StmntSeq ss)

= apsnd StmntSeq (cvm-fold ppStep (False,\_\text{com}) ss)

ppStmt (\(-,\_\text{com}\))

(StmntAtomic ss)

= apsnd StmntAtomic (cvm-fold ppStep (False,\_\text{com}) ss)

ppStmt (\(-,\_\text{com}\))

(StmntIf sss)

= apsnd StmntIf (cvm-fold (cvm-fold ppStep) (False,\_\text{com}) sss)

ppStmt (\(-,\_\text{com}\))

(StmntDo sss)

= apsnd StmntDo (cvm-fold (cvm-fold ppStep) (False,\_\text{com}) sss)

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntIncr v)

= ((False,ps,i,\_\text{com}), incr (liftChan (ppVarRef cvm v)))

ppStmt (\(-,ps,i,\_\text{com}\))

(StmntDecr v)

= ((False,ps,i,\_\text{com}), decr (liftChan (ppVarRef cvm v)))

ppStmt (\(-,\_\text{com}\))

(StmntPrintF - -)

22
with-warn "PrintF ignored" ((False, cvm), StmntSkip)
| ppStmnt (-, cm) (AST.StmntPrintM -)
  with-warn "PrintM ignored" ((False, cvm), StmntSkip)
| ppStmnt (-, ps, inl, cvm) (AST.StmntFor
  (AST.RangeFromTo i lb ub)
  steps) = (let
    i = liftChan (ppVarRef cvm i);
    (lb, ub) = (ppExpr cvm lb, ppExpr cvm ub)
in
  apsnd (forFromTo i lb ub) (cvm-fold ppStep (False, ps, inl, cvm) steps))
| ppStmnt (-, ps, inl, cvm) (AST.StmntFor
  (AST.RangeIn i v)
  steps) = (let
    i = liftChan (ppVarRef cvm i);
    (cvm', steps) = cvm-fold ppStep (False, ps, inl, cvm) steps
in
  case ppVarRef cvm v of
    Inr c ⇒ (cvm', forInChan i c steps)
  | Inl (VarRef - - (Some -)) ⇒ err "Iterating over array–member."
  | Inl (VarRef - name None) ⇒ (let (-, v, -) = cvm in
    case fst (the (lm.lookup name v)) of
    None ⇒ err "Iterating over non–array variable."
    | Some N ⇒ (cvm', forInArray i N steps)))
| ppStmnt (-, ps, inl, cvm) (AST.StmntSelect
  (AST.RangeFromTo i lb ub)) = (let
    i = liftChan (ppVarRef cvm i);
    (lb, ub) = (ppExpr cvm lb, ppExpr cvm ub)
in
  ((False, ps, inl, cvm), select i lb ub))
| ppStmnt (-, cm) (AST.StmntSelect (AST.RangeIn - -)) = err "in not allowed in select"
| ppStmnt (-, ps, inl, cvm) (AST.StmntCall macro args) = (let
  args = map (liftChan ◦ ppVarRef cvm) args;
  (c, v, m, a) = cvm
in
  case lm.lookup macro inl of
  None ⇒ errv "Calling unknown macro " macro
  | Some (names, sF) ⇒
    if length names ≠ length args then
      (err "Called macro with wrong number of arguments.")
    else

23
let a' = foldl (\(\lambda a (k,v)\). lm.update k v a) a (zip names args) in
let ((c,v,m,-),steps) = sF (c,v,m,a') in
((False,ps,nil,c,v,m,a), StmntSeq steps)

| ppStmnt cvm (AST.StmntDStep -) = use "StmntDStep"

fun ppModule
:: var-data \times inlines \Rightarrow AST.module
⇒ var-data \times inlines \times (varDecl list + proc + ltl)

where
ppModule (cvm, inl) (AST.ProcType act name args prio prov steps) = (let
   - = the-warn prio "Priorities for procs not supported. Ignored."
   - = the-warn prov "Provd (??) for procs not supported. Ignored."
   (cvm', args) = cvm-fold ppDeclProcArg cvm args
   ((-, vars, -, -), steps) = cvm-fold ppStep (True,\[],\[],cvm') steps in
   (cvm, inl, Inr (Inl (ProcType act name (concat args) vars steps))))

| ppModule (cvm, inl) (AST.Init prio steps) = (let
   - = the-warn prio "Priorities for procs not supported. Ignored."
   in
   let ((-, vars, -, -), steps) = cvm-fold ppStep (True,\[],\[],cvm) steps in
   (cvm, inl, Inr (Inl (Init vars steps))))

| ppModule (cvm, inl) (AST.Ltl name formula) =
   (cvm, inl, Inr (Inr (name, formula)))

| ppModule (cvm, inl) (AST.ModaDecl decl) =
   apsnd (\(\lambda ds. (\inl,\inl ds)\)) (ppDecl True cvm decl)

| ppModule (cvm, inl) (AST.MType mtys) = (let
   (c,v,m,a) = cvm in
   let num = integer-of-nat (lm.size m) + 1 in
   let (m',-) = foldr (\(\lambda mty (m,num)\).
   let m' = lm.update mty num m
   in (m',num+1)) mtys (m,num)
   in
   ((c,v,m',a), inl, Inl []))

| ppModule (cvm, inl) (AST.Inline name args steps) = (let
   stepF = (\(\lambda cvm. let ((-,-,-,cvm),steps) =
   cvm-fold ppStep (False,\[],\[],cvm) steps in
   (cvm,steps))
   in
   let inl = lm.update name (args, stepF) inl
   in
   (cvm,inl, Inl[]))

| ppModule cvm (AST.DProcType - - - -) = use "DProcType"
| ppModule cvm (AST.Never -) = use "Never"
| ppModule cvm (AST.Trace '-') = use "Trace"

24
| ppModule cvm (AST.NoTrace -) = use "NoTrace"
| ppModule cvm (AST.TypeDef -) = use "TypeDef"

**Definition**: `preprocess :: AST.module list ⇒ promela where`  
`preprocess ms = (`  
  `let`  
  `dflt-vars = [(STR "-pid", (None, False)),`  
  `(STR "--assert--", (None, True)),`  
  `(STR ".", (None, True))];`  
  `cvm = (lm.empty(), lm.to-map dflt-vars, lm.empty(), lm.empty());`  
  `(-,-,pr) = (foldl (λ(cvm, inl, vs, ps, ls) m.  
    `let (cvm', inl', m') = ppModule (cvm, inl) m in`  
    `case m' of`  
    `Inl v ⇒ (cvm', inl', vs@[v], ps, ls)  
    | Inr (Inl p) ⇒ (cvm', inl', vs, ps@[p], ls)  
    | Inr (Inr l) ⇒ (cvm', inl', vs, ps, ls@[l]))`  
    `(cvm, lm.empty(), [], [], []) ms)`  
  `in`  
  `pr)`

**Function**: `extractLTL :: AST.module ⇒ ltl option where`  
`extractLTL (AST.Ltl name formula) = Some (name, formula)`  
`| extractLTL _ = None`

**primrec extractLTLs :: AST.module list ⇒ (String.literal, String.literal) lm where**  
`extractLTLs [] = lm.empty()`  
`| extractLTLs (m#ms) = (case extractLTL m of`  
  `None ⇒ extractLTLs ms`  
  `| Some (n,f) ⇒ lm.update n f (extractLTLs ms))`

**Definition**: `lookupLTL :: AST.module list ⇒ String.literal ⇒ String.literal option where`  
`lookupLTL ast k = lm.lookup k (extractLTLs ast)`

### 3.3 The transition system

The edges in our transition system consist of a condition (evaluated under the current environment) and an effect (modifying the current environment). Further they may be atomic, i.e. a whole row of such edges is taken before yielding a new state. Additionally, they carry a priority: the edges are checked from highest to lowest priority, and if one edge on a higher level can be taken, the lower levels are ignored.

The states of the system do not carry any information.

**Datatype**: `edgeCond = EElse`
| ECTrue       |
| ECFalse      |
| ECEExpr expr |
| ECRun String.l literal |
| ECSend chanRef |
| ECRecv chanRef recvArg list bool |

datatype edgeEffect = EEEnd |
| EEId       |
| EEAssert expr |
| EEAssign varRef expr |
| EEDecl procVarDecl |
| EERun String.l literal expr list |
| EESend chanRef expr list bool |
| EERecv chanRef recvArg list bool bool |

datatype edgeIndex = Index nat | LabelJump String.l literal nat option |

datatype edgeAtomic = NonAtomic | Atomic | InAtomic |

record edge = |
| cond :: edgeCond |
| effect :: edgeEffect |
| target :: edgeIndex |
| prio :: integer |
| atomic :: edgeAtomic |

definition isAtomic :: edge ⇒ bool where |
| isAtomic e = (case atomic e of Atomic ⇒ True | - ⇒ False) |

definition inAtomic :: edge ⇒ bool where |
| inAtomic e = (case atomic e of NonAtomic ⇒ False | - ⇒ True) |

3.4 State |

datatype variable = Var varType integer |
| VArray varType nat integer iarray |

datatype channel = Channel integer varType list integer list list |
| HSCChannel varType list |
| InvChannel |

type-synonym var-dict = (String.l literal, variable) lm |

type-synonym labels = (String.l literal, nat) lm |

type-synonym lls = (String.l literal, String.l literal) lm |

type-synonym states = (integer (*prio*) × edge list) iarray |

type-synonym channels = channel list |

type-synonym process =

26
nat (*offset*)
x edgeIndex (*start*)
x procArg list (*args*)
x varDecl list (*top decls *)

record program =
processes :: process iarray
labels :: labels iarray
states :: states iarray
proc-names :: String.literal iarray
proc-data :: (String.literal, nat) lm

record pState = — State of a process
pid :: nat — Process identifier
vars :: var-dict — Dictionary of variables
pc :: nat — Program counter
channels :: integer list — List of channels created in the process. Used to close
them on finalization.
idx :: nat — Offset into the arrays of program

hide-const (open) idx

record gState = — Global state
vars :: var-dict — Global variables
channels :: channels — Channels are by construction part of the global state,
even when created in a process.
timeout :: bool — Set to True if no process can take a transition.
procs :: pState list — List of all running processes. A process is removed from
it, when there is no running one with a higher index.

record gState1 = gState + — Additional internal infos
handshake :: nat
hsdata :: integer list — Data transferred via a handshake.
exclusive :: nat — Set to the PID of the process, which is in an exclusive (=
atomic) state.
else :: bool — Set to True for each process, if it can not take a transition.
Used before timeout.

3.5 Printing

primrec printBinOp :: binOp ⇒ string where
  printBinOp BinOpAdd = "+"
| printBinOp BinOpSub = "−"
| printBinOp BinOpMul = "∗"
| printBinOp BinOpDiv = "/"
| printBinOp BinOpMod = "mod"
| printBinOp BinOpGr = "<"
| printBinOp BinOpLe = "≤"
| printBinOp BinOpGEq = "≥"
primrec printUnOp :: unOp ⇒ string where
printUnOp UnOpMinus = "−"
printUnOp UnOpNeg = "!"

definition printList :: ('a ⇒ string) ⇒ 'a list ⇒ string ⇒ string ⇒ string ⇒ string ⇒ string where
printList f xs l r sep = (let f' = (λstr x. if str = [] then f x else str @ sep @ f x)
in l @ (foldl f' [] xs) @ r)

lemma printList-cong [fundef-cong]:
assumes xs = xs'
and l = l'
and r = r'
and sep = sep'
and ∀x. x ∈ set xs ⇒ f x = f' x
shows printList f xs l r sep = printList f' xs' l' r' sep'
(proof)

fun printExpr :: (integer ⇒ string) ⇒ expr ⇒ string
and printFun :: (integer ⇒ string) ⇒ string ⇒ chanRef ⇒ string
and printVarRef :: (integer ⇒ string) ⇒ varRef ⇒ string
and printChanRef :: (integer ⇒ string) ⇒ chanRef ⇒ string
and printrecvArg :: (integer ⇒ string) ⇒ recvArg ⇒ string where
printExpr f ExprTimeOut = "timeout"
| printExpr f (ExprBinOp binOp left right) =
  printExpr f left @ "" @ printBinOp binOp @ "" @ printExpr f right
| printExpr f (ExprUnOp unOp e) = printUnOp unOp @ printExpr f e
| printExpr f (ExprVarRef varRef) = printVarRef f varRef
| printExpr f (ExprConst i) = f i
| printExpr f (ExprMConst i m) = Stringexplode m
| printExpr f (ExprCond e c l r) =
  "(" @ printExpr f e @ ")" -> ""
  @ printExpr f l @ "";
  @ printExpr f r @ "")""
| printExpr f (ExprLen chan) = printFun f "len" chan
| printExpr f (ExprEmpty chan) = printFun f "empty" chan
| printExpr f (ExprFull chan) = printFun f "full" chan
| printExpr f (ExprPoll chan es srt) = (let p = if srt then "???" else "??" in
  printChanRef f chan @ p)
printList(printRecvArg f) es ['""' @ printExpr f indx @ ';' @ '""]

printVarRef - (VarRef - name None) = String.explode name
printVarRef f (VarRef - name (Some indx)) =
  String.explode name @ '[' @ printExpr f indx @ ']'

printChanRef f (ChanRef v) = printVarRef f v

printFun f fun var = fun @ '(' @ printChanRef f var @ ')'

printVarDecl :: procVarDecl => string where
printVarDecl f (ProcVarDeclNum - n None None) =
  String.explode n @ '=' @ '0'
printVarDecl f (ProcVarDeclNum - n None (Some e)) =
  String.explode n @ '=' @ printExpr f e
printVarDecl f (ProcVarDeclNum - n (Some i) None) =
  String.explode n @ '[' @ f i @ ']' = @ '0'
printVarDecl f (ProcVarDeclNum - n (Some i) (Some e)) =
  String.explode n @ '[' @ f i @ ']' = @ printExpr f e
printVarDecl f (ProcVarDeclChan n None) =
  "chan " @ String.explode n
printVarDecl f (ProcVarDeclChan n (Some i)) =
  "chan " @ String.explode n @ '[' @ f i @ ']'"

primrec printCond :: edgeCond => string where
printCond f ECElse = "else"
printCond f ECTrue = "true"
printCond f ECFalse = "false"
p
printCond f (ECRun n) = "run " @ String.explode n @ "(...)"
p
printCond f (ECEExpr e) = printExpr f e

primrec printEffect :: edgeEffect => string where
printEffect f EEEnd = "-- end --"
printEffect f EEId = "ID"

printEffect f EEAssign v expr =
  printVarRef f v @ '=' @ printExpr f expr

primrec printEnv f (EEDecl d) = printVarDecl f d

let s = if srt then '!!' else '!' in
printEdges definition | printProcesses
primrec printIndex :: (integer ⇒ string) ⇒ edgeIndex ⇒ string where
printIndex f (Index pos) = f (integer-of-nat pos)
| printIndex - (LabelJump l _) = String.explode l

definition printEdge :: (integer ⇒ string) ⇒ nat ⇒ edge ⇒ string where
printEdge f indx e = (let
  tStr = printIndex f (target e);
  pStr = if prio e < 0 then "Prio: " @ f (prio e) else [];
  atom = if isAtomic e then λx. x @ "\{A\}" else id;
  pEff = λ-.. atom (printEffect f (effect e));
  contStr = case (cond e) of
    ECTTrue ⇒ pEff ()
    ECFalse ⇒ pEff ()
    ECSend ⇒ pEff()
    ECREc - - ⇒ pEff()
    _ ⇒ atom ("((" @ printCond f (cond e) @ "))")
  in
  f (integer-of-natindx) @ "-" @ tStr @ "@" @ pStr)

definition printEdges :: (integer ⇒ string) ⇒ states ⇒ string list where
printEdges f es = concat (map (λn. map (printEdge f n) (snd (es !! n))))
  (rev [0..<IArray.length es]))

definition printLabels :: (integer ⇒ string) ⇒ labels ⇒ string list where
printLabels f ls = lm.iterate ls (λ(k,l) res.
  ("Label " @ String.explode k @ ": "
  @ f (integer-of-nat l)) # res) []

fun printProcesses :: (integer ⇒ string) ⇒ program ⇒ string list where
printProcesses f prog = lm.iterate (proc-data prog)
  (λ(k,idx) res.
    let (-,start,..) = processes prog !! idx in
    [] # ("Process " @ String.explode k) # [] # printEdges f (states prog !!
    idx)
      @ ["START ----> " @ printIndex f start, []]
      @ printLabels f (labels prog !! idx) @ res] []
begin

The different data structures used in the Promela implementation require different invariants, which are specified in this file. As there is no (useful) way of specifying correctness of the implementation, those invariants are tailored towards proving the finiteness of the generated state-space.

4.1 Bounds

Finiteness requires that possible variable ranges are finite, as is the maximum number of processes. Currently, they are supplied here as constants. In a perfect world, they should be able to be set dynamically.

definition min-var-value :: integer where
min-var-value = -(2^31)
definition max-var-value :: integer where
max-var-value = (2^31) - 1

lemma min-max-var-value-simps [simp, intro!]:
min-var-value < max-var-value
min-var-value < 0
min-var-value ≤ 0
max-var-value > 0
max-var-value ≥ 0

definition max-procs ≡ 255
definition max-channels ≡ 65535
definition max-array-size = 65535

4.2 Variables and similar

fun varType-inv :: varType ⇒ bool where
varType-inv (VTBounded l h)
←→ l ≥ min-var-value ∧ h ≤ max-var-value ∧ l < h
| varType-inv VTChan ←→ True

fun variable-inv :: variable ⇒ bool where
variable-inv (Var t val)
←→ varType-inv t ∧ val ∈ {min-var-value..max-var-value}
| variable-inv (VArray t sz ar)
←→ varType-inv t ∧ sz ≤ max-array-size ∧ IArray.length ar = sz ∧ set (IArray.list-of ar) ⊆ {min-var-value..max-var-value}

fun channel-inv :: channel ⇒ bool where
channel-inv (Channel cap ts q)
\[ \text{cap} \leq \text{max-array-size} \]
\[ \land \text{cap} \geq 0 \]
\[ \land \text{set } ts \subseteq \text{Collect varType-inv} \]
\[ \land \text{length } ts \leq \text{max-array-size} \]
\[ \land \text{length } q \leq \text{max-array-size} \]
\[ \land (\forall x \in \text{set } q. \text{length } x = \text{length } ts) \]
\[ \land \text{set } x \subseteq \{\text{min-var-value..max-var-value}\} \]
\[ \text{channel-inv} (\text{HSChannel } ts) \]
\[ \land \text{set } ts \subseteq \text{Collect varType-inv} \land \text{length } ts \leq \text{max-array-size} \]
\[ \text{channel-inv InvChannel} \iff \text{True} \]

**Lemma** varTypes-finite:
\[ \text{finite} (\text{Collect varType-inv}) \]
\[ \langle \text{proof} \rangle \]

**Lemma** variables-finite:
\[ \text{finite} (\text{Collect variable-inv}) \]
\[ \langle \text{proof} \rangle \]

**Lemma** channels-finite:
\[ \text{finite} (\text{Collect channel-inv}) \]
\[ \langle \text{proof} \rangle \]

To give an upper bound of variable names, we need a way to calculate it.

**Primrec** procArgName :: procArg \Rightarrow \text{String.literal} \text{ where}
procArgName (ProcArg - name) = name

**Primrec** varDeclName :: varDecl \Rightarrow \text{String.literal} \text{ where}
varDeclName (VarDeclNum - - name - -) = name
| varDeclName (VarDeclChan name - -) = name

**Primrec** procVarDeclName :: procVarDecl \Rightarrow \text{String.literal} \text{ where}
procVarDeclName (ProcVarDeclNum - - name - -) = name
| procVarDeclName (ProcVarDeclChan name - -) = name

**Definition** edgeDecls :: edge \Rightarrow \text{procVarDecl set} \text{ where}
edgeDecls e = (\text{case effect } e \text{ of}
EEDecl p \Rightarrow \{p\}
| - \Rightarrow \{\})

**Lemma** edgeDecls-finite:
\[ \text{finite} (\text{edgeDecls } e) \]
\[ \langle \text{proof} \rangle \]

**Definition** edgeSet :: states \Rightarrow \text{edge set} \text{ where}
edgeSet s = set (concat (map snd (IArray.list-of s)))

**Lemma** edgeSet-finite:
finite (\text{edgeSet} \; s)
\langle \text{proof} \rangle

definition \text{statesDecls} :: \text{states} \Rightarrow \text{procVarDecl} \; \text{set} \quad \text{where} \\
\text{statesDecls} \; s = \text{UNION} \; (\text{edgeSet} \; s) \; \text{edgeDecls}

definition \text{statesNames} :: \text{states} \Rightarrow \text{String.l literal set} \quad \text{where}
\text{statesNames} \; s = \text{procVarDeclName} \; \cdot \; \text{statesDecls} \; s

lemma \text{statesNames-finite}: 
finite \; (\text{statesNames} \; s)
\langle \text{proof} \rangle

fun \text{process-names} :: \text{states} \Rightarrow \text{process} \Rightarrow \text{String.l literal set} \quad \text{where}
\text{process-names} \; ss \; (\cdot, \; \cdot, \; \text{args, decls}) = \\
\text{statesNames} \; ss \\
\cup \; \text{procArgName} \; \cdot \; \text{set args} \\
\cup \; \text{varDeclName} \; \cdot \; \text{set decls} \\
\cup \{\text{STR ""-", STR ""-assert--", STR ""-pid""}\}

lemma \text{process-names-finite}: 
finite \; (\text{process-names} \; ss \; p)
\langle \text{proof} \rangle

definition \text{vardict-inv} :: \text{states} \Rightarrow \text{process} \Rightarrow \text{var-dict} \Rightarrow \text{bool} \quad \text{where}
\text{vardict-inv} \; ss \; p \; vs \; \longleftrightarrow \\
\text{lm. to-map} \; \text{vs} \; (\lambda \; (k, \; v). \; k \in \text{process-names} \; ss \; p \land \text{variable-inv} \; v)

lemma \text{vardicts-finite}: 
finite \; (\text{Collect} \; (\text{vardict-inv} \; ss \; p))
\langle \text{proof} \rangle

lemma \text{lm-to-map-vardict-inv}: 
\text{assumes} \; \forall \; (k, \; v) \in \text{set} \; xs. \; k \in \text{process-names} \; ss \; p \land \text{variable-inv} \; v \\
\text{shows} \; \text{vardict-inv} \; ss \; p \; (\text{lm. to-map} \; xs)
\langle \text{proof} \rangle

4.3 Invariants of a process

definition \text{pState-inv} :: \text{program} \Rightarrow \text{pState} \Rightarrow \text{bool} \quad \text{where}
\text{pState-inv} \; \text{prog} \; p \\
\longleftrightarrow \; \text{pid} \; p \leq \text{max-procs} \\
\land \; \text{pState.idx} \; p < \text{IArray.length} \; (\text{states} \; \text{prog}) \\
\land \; \text{IArray.length} \; (\text{states} \; \text{prog}) = \text{IArray.length} \; (\text{processes} \; \text{prog}) \\
\land \; \text{pc} \; p < \text{IArray.length} \; ((\text{states} \; \text{prog}) \; \text{!!} \; \text{pState.idx} \; p) \\
\land \; \text{set} \; (\text{pState.channels} \; p) \subseteq \{-1..<\text{integer-of-nat} \; \text{max-channels}\} \\
\land \; \text{length} \; (\text{pState.channels} \; p) \leq \text{max-channels} \\
\land \; \text{vardict-inv} \; ((\text{states} \; \text{prog}) \; \text{!!} \; \text{pState.idx} \; p)
lemma pStates-finite:
finite (Collect (pState-inv prog))
⟨proof⟩

Throughout the calculation of the semantic engine, a modified process is not necessarily part of \textit{procs g}. Hence we need to establish an additional constraint for the relation between a global and a process state.

definition cl-inv :: ('a gState-scheme * pState) ⇒ bool where
cl-inv gp = (case gp of (g,p) ⇒
length (pState.channels p) ≤ length (gState.channels g))

lemma cl-inv-lengthD:
cl-inv (g,p) ⇒ length (pState.channels p) ≤ length (gState.channels g)
⟨proof⟩

lemma cl-invI:
length (pState.channels p) ≤ length (gState.channels g) ⇒ cl-inv (g,p)
⟨proof⟩

lemma cl-inv-trans:
length (channels g) ≤ length (channels g') ⇒ cl-inv (g,p) ⇒ cl-inv (g',p)
⟨proof⟩

lemma cl-inv-vars-update[intro!]:
cl-inv (g,p) ⇒ cl-inv (g, pState-vars-update vs p)
cl-inv (g,p) ⇒ cl-inv (gState-vars-update vs g, p)
⟨proof⟩

lemma cl-inv-handshake-update[intro!]:
cl-inv (g,p) ⇒ cl-inv (g\{handshake := h\},p)
⟨proof⟩

lemma cl-inv-hsdata-update[intro!]:
cl-inv (g,p) ⇒ cl-inv (g\{hsdata := h\},p)
⟨proof⟩

lemma cl-inv-procs-update[intro!]:
cl-inv (g,p) ⇒ cl-inv (g\{procs := ps\},p)
⟨proof⟩

lemma cl-inv-channels-update:
assumes cl-inv (g,p)
shows cl-inv (gState-channels-update (λcs. cs[i:=c]) g, p)
⟨proof⟩
4.4 Invariants of the global state

Note that $gState-inv$ must be defined in a way to be applicable to both $gState$ and $gStateI$.

**Definition** $gState-inv$ :: program $\Rightarrow$ (a gState-scheme $\Rightarrow$ bool) where

\[
gState-inv prog g \iff 
\begin{align*}
& \text{length (procs } g) \leq \text{max-procs} \\
& (\forall p \in \text{set (procs } g). \ pState-inv prog p \land \cl-inv (g,p)) \\
& \land \ \text{set (channels } g) \subseteq \text{Collect channel-inv} \\
& \land \ \text{lm.ball (vars } g) (\lambda(k,v). \ \text{variable-inv } v)
\end{align*}
\]

The set of global states adhering to the terms of $gState-inv$ is not finite. But the set of all global states that can be constructed by the semantic engine from one starting state is. Thus we establish a progress relation, i.e. all successors of a state $g$ relate to $g$ under this specification.

**Definition** $gState-progression-rel$ :: program $\Rightarrow$ (a gState-scheme) rel where

\[
gState-progression-rel p = \{(g,g'). \ gState-inv p g \land gState-inv p g' \\
\land \ \text{length (channels } g) \leq \text{length (channels } g') \\
\land \ \text{dom (lm.}\alpha\text{ (vars } g)) = \text{dom (lm.}\alpha\text{ (vars } g'))\}
\]

**Lemma** $gState-progression-rel-gState-invI1$ [intro]:

\[
(g,g') \in gState-progression-rel prog \Longrightarrow gState-inv prog g
\]

**Lemma** $gState-progression-rel-gState-invI2$ [intro]:

\[
(g,g') \in gState-progression-rel prog \Longrightarrow gState-inv prog g'
\]

**Lemma** $gState-progression-reflI$:

assumes $gState-inv prog g$

and $gState-inv prog g'$

and $\text{length (channels } g) \leq \text{length (channels } g')$

and $\text{dom (lm.}\alpha\text{ (vars } g)) = \text{dom (lm.}\alpha\text{ (vars } g'))$

shows $(g,g') \in gState-progression-rel prog$

**Lemma** $gState-progression-refl[simp,intro!]$:

\[
gState-inv prog g \Longrightarrow (g,g) \in (gState-progression-rel prog)
\]

**Lemma** refl-on-gState-progression-rel:

refl-on (Collect (gState-inv prog)) (gState-progression-rel prog)

**Lemma** trans-gState-progression-rel[simp]:

trans (gState-progress-rel prog)
lemmas $g\text{State}\text{-progress-rel-trans}$ [trans] = $\text{trans}\text{-}g\text{State}\text{-progress-rel} [\text{THEN} \text{transD}]$

lemma $g\text{State}\text{-progress-rel-trancl-id}$ [simp]:
\[(g\text{State}\text{-progress-rel} \text{prog})^+ = g\text{State}\text{-progress-rel} \text{prog}\]
\(\langle \text{proof} \rangle\)

lemma $g\text{State}\text{-progress-rel-rtrancl-absorb}$:
\begin{align*}
\text{assumes} & \quad g\text{State}-\text{inv} \text{ prog} \ g \\
\text{shows} & \quad (g\text{State}\text{-progress-rel} \text{prog})^+ \{g\} = g\text{State}\text{-progress-rel} \text{prog} \{g\} \\
\langle \text{proof} \rangle
\end{align*}

The main theorem: The set of all global states reachable from an initial state, is finite.

lemma $g\text{States-finite}$:
\begin{align*}
\text{fixes} & \quad g :: g\text{State} \\
\text{shows} & \quad \text{finite} ((g\text{State}\text{-progress-rel} \text{prog})^+ \{g\}) \\
\langle \text{proof} \rangle
\end{align*}

lemma $g\text{State}\text{-progress-rel-channels-update}$:
\begin{align*}
\text{assumes} & \quad g\text{State}-\text{inv} \text{ prog} \ g \\
& \quad \text{and} \ channel-\text{inv} c \\
& \quad \text{and} \ i < \text{length} (\text{channels} \ g) \\
\text{shows} & \quad (g, g\text{State}.\text{channels-update} (\lambda \text{cs}. \text{cs}[i:=c]) \ g) \in g\text{State}\text{-progress-rel} \text{prog} \\
\langle \text{proof} \rangle
\end{align*}

lemma $g\text{State}\text{-progress-rel-channels-update-step}$:
\begin{align*}
\text{assumes} & \quad g\text{State}-\text{inv} \text{ prog} \ g \\
& \quad \text{and} \ \text{step} : (g, g') \in g\text{State}\text{-progress-rel} \text{prog} \\
& \quad \text{and} \ channel-\text{inv} c \\
& \quad \text{and} \ i < \text{length} (\text{channels} \ g') \\
\text{shows} & \quad (g, g\text{State}.\text{channels-update} (\lambda \text{cs}. \text{cs}[i:=c]) \ g') \in g\text{State}\text{-progress-rel} \text{prog} \\
\langle \text{proof} \rangle
\end{align*}

4.5 Invariants of the program

Naturally, we need our program to also adhere to certain invariants. Else we can’t show, that the generated states are correct according to the invariants above.

definition $\text{program-inv}$ where
\begin{align*}
\text{program-inv} \text{ prog} & \iff I\text{Array}.\text{length} (\text{states} \text{ prog}) > 0 \\
& \quad \land I\text{Array}.\text{length} (\text{states} \text{ prog}) = I\text{Array}.\text{length} (\text{processes} \text{ prog}) \\
& \quad \land (\forall s \in \text{set} (I\text{Array}.\text{list-of} (\text{states} \text{ prog})). I\text{Array}.\text{length} s > 0) \\
& \quad \land \text{lm}\text{-ball} (\text{proc-data} \text{ prog}) \\
& \quad \quad (\lambda (s, sidx). \\
& \quad \quad \quad sidx < I\text{Array}.\text{length} (\text{processes} \text{ prog}) \\
& \quad \quad \quad \land \text{fst} (\text{processes} \text{ prog} ! sidx) = \text{sidx})
\end{align*}
\( \forall (\text{sidx, start, procArgs, args}) \in \text{set (IArray.list-of (processes prog))}. \\
(\exists s. \text{start} = \text{Index s} \land s < \text{IArray.length (states prog !! sidx)}) \)

lemma program-inv-length-states:
  assumes program-inv prog
  and \( n < \text{IArray.length (states prog)} \)
  shows \( \text{IArray.length (states prog !! n)} > 0 \)
  ⟨proof⟩

lemma program-invI:
  assumes \( 0 < \text{IArray.length (states prog)} \)
  and \( \text{IArray.length (states prog)} = \text{IArray.length (processes prog)} \)
  and \( \forall s. s \in \text{set (IArray.list-of (states prog))} \)
    \( \implies 0 < \text{IArray.length s} \)
  and \( \forall \text{sidx. sidx} \in \text{ran (lm.\alpha (proc-data prog))} \)
    \( \implies \text{sidx} < \text{IArray.length (processes prog)} \)
    \( \land \text{fst (processes prog !! sidx)} = \text{sidx} \)
  and \( \forall \text{sidx start procArgs args.} \)
    \( (\text{sidx,start,procArgs, args}) \in \text{set (IArray.list-of (processes prog))} \)
    \( \implies \exists s. \text{start} = \text{Index s} \land s < \text{IArray.length (states prog !! sidx)} \)
  shows program-inv prog
  ⟨proof⟩

end

5 Formalization of Promela semantics

theory Promela
imports
  PromelaDatastructures
  PromelaInvariants
  PromelaStatistics
begin
After having defined the datastructures, we present in this theory how to construct the transition system and how to generate the successors of a state, i.e. the real semantics of a Promela program. For the first task, we take the enriched AST as input, the second one operates on the transition system.

5.1 Misc Helpers
definition add-label :: String.literal \(\Rightarrow\) labels \(\Rightarrow\) nat \(\Rightarrow\) labels where
  add-label l lbls pos = ( 
    case \text{lm.lookup} l lbls of 
      None \Rightarrow \text{lm.update} l \text{pos} lbls 
    \mid \text{Some} - \Rightarrow \text{abortv} "Label given twice: " l (\lambda. lbls)

37
definition \texttt{min-prio} :: \texttt{edge list} $\Rightarrow$ \texttt{integer} $\Rightarrow$ \texttt{integer} \ where \\
\hspace{1em} \texttt{min-prio es start} = \texttt{Min} ((\texttt{prio} \ ' \set es) \cup \{\texttt{start}\})

lemma \texttt{min-prio-code} \ [\texttt{code}]:: \\
\hspace{1em} \texttt{min-prio es start} = \texttt{fold} (\lambda e \texttt{pri}. \texttt{if prior e < prior e else pri}) \texttt{es start} \hspace{1em} \langle \texttt{proof} \rangle

definition \texttt{for-all} :: ('a $\Rightarrow$ \texttt{bool}) $\Rightarrow$ '\a list $\Rightarrow$ \texttt{bool} \ where \\
\hspace{1em} \texttt{for-all f xs} $\longleftrightarrow$ (\forall x \in \set{x}s. f x)

lemma \texttt{for-all-code} \ [\texttt{code}]:: \\
\hspace{1em} \texttt{for-all f xs} $\longleftrightarrow$ \texttt{foldli xs id (\lambda kv. f kv) True} \hspace{1em} \langle \texttt{proof} \rangle

definition \texttt{find-remove} :: ('a $\Rightarrow$ \texttt{bool}) $\Rightarrow$ '\a list $\Rightarrow$ '\a option $\times$ '\a list \ where \\
\hspace{1em} \texttt{find-remove P xs} = (\texttt{case List.find P xs of None $\Rightarrow$ (None, xs)} \\
\hspace{2em} | Some x $\Rightarrow$ (Some x, List.remove1 x xs))

lemma \texttt{find-remove-code} \ [\texttt{code}]:: \\
\hspace{1em} \texttt{find-remove P \[]} = (\texttt{None, \[]}) \\
\hspace{1em} \texttt{find-remove P (x#xs)} = (\texttt{if P x then (Some x, xs)} \\
\hspace{3em} else \texttt{apsnd (Cons x) (find-remove P xs)}) \hspace{1em} \langle \texttt{proof} \rangle

lemma \texttt{find-remove-subset}:: \\
\hspace{1em} \texttt{find-remove P xs} = (\texttt{res, xs')} $\Longrightarrow$ \texttt{set xs'} $\subseteq$ \texttt{set xs} \hspace{1em} \langle \texttt{proof} \rangle

lemma \texttt{find-remove-length}:: \\
\hspace{1em} \texttt{find-remove P xs} = (\texttt{res, xs'}) $\Longrightarrow$ \texttt{length xs'} $\leq$ \texttt{length xs} \hspace{1em} \langle \texttt{proof} \rangle

5.2 Variable handling

Handling variables, with their different scopes (global vs. local), and their different types (array vs channel vs bounded) is one of the main challenges of the implementation.

fun \texttt{lookupVar} :: \texttt{variable} $\Rightarrow$ \texttt{integer option} $\Rightarrow$ \texttt{integer} \ where \\
\hspace{1em} \texttt{lookupVar (Var - val) None} = val \\
\hspace{2em} | \texttt{lookupVar (Var - -) (Some -)} = \texttt{abort "Array used on var" (\lambda..0)} \\
\hspace{2em} | \texttt{lookupVar (VArray - - vals) None} = vals !! 0 \\
\hspace{2em} | \texttt{lookupVar (VArray - siz vals) (Some idx)} = vals !! \texttt{nat-of-integer idx}

primrec \texttt{checkVarValue} :: \texttt{varType} $\Rightarrow$ \texttt{integer} $\Rightarrow$ \texttt{integer} \ where \\
\hspace{1em} \texttt{checkVarValue (VTBounded lRange hRange) val} = ( \\
\hspace{2em} if val $\leq$ hRange $\land$ val $\geq$ lRange then val \\
\hspace{3em} else (overflowing is well-defined and may actually be used (e.g. bool) *) \\
\hspace{4em} if hRange = 0 $\land$ val $> 0$ \\
\hspace{5em} then val mod (hRange + 1)
else (* we do not want to implement C−semantics (ie type casts) *)
  abort "Value overflow" (λ- lRange))
  | checkVarValue VTChan val = (if val < min-var-value ∨ val > max-var-value
  then abort "Value overflow" (λ- 0)
  else val)

lemma [simp]:
  variable-inv (Var VTChan 0)
⟨proof⟩

lemma checkVarValue-bounds:
  varType-inv type ⇒ checkVarValue type val ≤ max-var-value
  varType-inv type ⇒ min-var-value ≤ checkVarValue type val
⟨proof⟩

lemma checkVarValue-Var:
  varType-inv type ⇒ variable-inv (Var type (checkVarValue type val))
⟨proof⟩

fun editVar :: variable ⇒ integer option ⇒ integer ⇒ variable where
  editVar (Var type -) None val = Var type (checkVarValue type val)
  | editVar (Var -) (Some -) - = abort "Array used on var" (λ- Var VTChan 0)
  | editVar (VArray type siz vals) None val = (let lv = IArray.list-of vals in
     let v′ = lv[0:=checkVarValue type val] in
     VArray type siz (IArray v′))
  | editVar (VArray type siz vals) (Some idx) val = (let lv = IArray.list-of vals in
     let v′ = lv[(nat-of-integer idx):=checkVarValue type val] in
     VArray type siz (IArray v′))

lemma editVar-variable-inv:
  assumes variable-inv v
  shows variable-inv (editVar v idx val)
⟨proof⟩

definition getVar' :: bool ⇒ String.literal ⇒ integer option
  ⇒ 'a gState-scheme ⇒ pState
  ⇒ integer option
where
  getVar' gl v idx g p = (let vars = if gl then gState.vars g else pState.vars p in
     map-option (λx. lookupVar x idz) (lm.lookup v vars))

definition setVar' :: bool ⇒ String.literal ⇒ integer option
  ⇒ integer
\[ \Rightarrow 'a \text{gState-scheme} \Rightarrow p\text{State} \]
\[ \Rightarrow 'a \text{gState-scheme} * p\text{State} \]

where

\[ \text{setVar}' \ gl \ v \ idx \ val \ g \ p = ( \]
\[ \quad \text{if gl then} \]
\[ \quad \quad \text{if v = STR } ' - ' \text{ then } (g,p) \) (* '-' is a write-only scratch variable *)
\[ \quad \quad \text{else case lm.lookup v (gState.vars g) of} \]
\[ \quad \quad \quad \text{None } \Rightarrow \text{abortv } ' ' \text{Unknown global variable: } v (\lambda \cdot (g,p)) \]
\[ \quad \quad \quad \mid \text{Some } x \Rightarrow (g(\text{gState.vars := lm.update v (editVar x idx val)}) \)
\[ \quad \quad \quad \quad (gState.gvars g)) \]
\[ \quad \quad \quad , p) \]
\[ \quad \text{else} \]
\[ \quad \quad \text{case lm.lookup v (pState.vars p) of} \]
\[ \quad \quad \quad \text{None } \Rightarrow \text{abortv } ' ' \text{Unknown proc variable: } v (\lambda \cdot (g,p)) \]
\[ \quad \quad \quad | \text{Some } x \Rightarrow (g, p(pState.vars := lm.update v (editVar x idx val)}) \)
\[ \quad \quad \quad (pState.vars p))) \]

lemma setVar'-gState-inv:
\[ \text{assumes gState-inv prog g} \]
\[ \text{shows gState-inv prog (fst (setVar' gl v idx val g p))} \]
\[ \langle \text{proof} \rangle \]

lemma setVar'-gState-progress-rel:
\[ \text{assumes gState-inv prog g} \]
\[ \text{shows } (g, \text{fst (setVar' gl v idx val g p)}) \in \text{gState-progress-rel prog} \]
\[ \langle \text{proof} \rangle \]

lemma vardict-inv-process-names:
\[ \text{assumes vardict-inv ss proc v} \]
\[ \text{and lm.lookup k v = Some x} \]
\[ \text{shows k \in process-names ss proc} \]
\[ \langle \text{proof} \rangle \]

lemma vardict-inv-variable-inv:
\[ \text{assumes vardict-inv ss proc v} \]
\[ \text{and lm.lookup k v = Some x} \]
\[ \text{shows variable-inv x} \]
\[ \langle \text{proof} \rangle \]

lemma vardict-inv-updateI:
\[ \text{assumes vardict-inv ss proc vs} \]
\[ \text{and } x \in \text{process-names ss proc} \]
\[ \text{and variable-inv v} \]
\[ \text{shows vardict-inv ss proc (lm.update x v vs)} \]
\[ \langle \text{proof} \rangle \]

lemma update-vardict-inv:
\[ \text{assumes vardict-inv ss proc v} \]
\[ \text{and lm.lookup k v = Some x} \]
and variable-inv \(x'\)  
shows vardict-inv ss proc (\(\text{lm.update } k \ x' \ v\))  
\(\langle \text{proof}\rangle\)

\textbf{lemma} setVar'-pState-inv:  
assumes pState-inv prog p  
shows pState-inv prog (snd (setVar' gl v idx val g p))  
\(\langle \text{proof}\rangle\)

\textbf{lemma} setVar'-cl-inv:  
assumes cl-inv (g,p)  
shows cl-inv (setVar' gl v idx val g p)  
\(\langle \text{proof}\rangle\)

\textbf{definition} withVar':: bool \Rightarrow String.literal \Rightarrow integer option  
\Rightarrow (integer \Rightarrow 'x)  
\Rightarrow 'a gState-scheme \Rightarrow pState  
\Rightarrow 'x  
\textbf{where}  
withVar' gl v idx f g p = f (the (getVar' gl v idx g p))

\textbf{definition} withChannel':: bool \Rightarrow String.literal \Rightarrow integer option  
\Rightarrow (nat \Rightarrow channel \Rightarrow 'x)  
\Rightarrow 'a gState-scheme \Rightarrow pState  
\Rightarrow 'x  
\textbf{where}  
withChannel' gl v idx f g p = (  
let error = \(\lambda\). abortv "Variable is not a channel: " v  
(\(\lambda\). f 0 InvChannel) in  
let abort = \(\lambda\). abortv "Channel already closed / invalid: " v  
(\(\lambda\). f 0 InvChannel)  
in withVar' gl v idx (\(\lambda i\). let i = nat-of-integer i in  
if \(i \geq \text{length } (\text{channels } g)\) then error ()  
else let c = channels g ! i in  
case c of  
InvChannel \Rightarrow abort ()  
| \_ \Rightarrow f i c) g p)

5.3 Expressions

Expressions are free of side-effects.  
This is in difference to SPIN, where run is an expression with side-effect.  
We treat run as a statement.

\textbf{abbreviation} trivCond x \equiv \text{if } x \text{ then } 1 \text{ else } 0

\textbf{fun} exprArith :: 'a gState-scheme \Rightarrow pState \Rightarrow expr \Rightarrow integer
\textbf{and} \texttt{pollCheck} :: 'a gState-scheme ⇒ pState ⇒ channel ⇒ recvArg list ⇒ bool ⇒ bool

\textbf{and} \texttt{recvArgsCheck} :: 'a gState-scheme ⇒ pState ⇒ recvArg list ⇒ integer list ⇒ bool

\textbf{where}

\texttt{exprArith} g p (ExprConst x) = x

\texttt{exprArith} g p (ExprMConst x _) = x

\texttt{exprArith} g p ExprTimeOut = \texttt{trivCond} (timeout g)

\texttt{exprArith} g p (ExprLen (ChanRef (VarRef gl name None))) =
\begin{center}
withChannel' gl name None ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} - - q \Rightarrow \text{integer-of-nat} (\text{length} q) \\
\quad | \text{HSChannel} - \Rightarrow 0) \ g p
\end{center}

\texttt{exprArith} g p (ExprLen (ChanRef (VarRef gl name (Some idx))) =
\begin{center}
withChannel' gl name (Some (exprArith g p idx)) ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} - - q \Rightarrow \text{integer-of-nat} (\text{length} q) \\
\quad | \text{HSChannel} - \Rightarrow 0) \ g p
\end{center}

\texttt{exprArith} g p (ExprEmpty (ChanRef (VarRef gl name None))) =
\begin{center}
\texttt{trivCond} (withChannel' gl name None ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} - - q \Rightarrow (q = [])) \\
\quad | \text{HSChannel} - \Rightarrow \text{True}) \ g p
\end{center}

\texttt{exprArith} g p (ExprEmpty (ChanRef (VarRef gl name (Some idx))) =
\begin{center}
\texttt{trivCond} (withChannel' gl name (Some (exprArith g p idx)) ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} - - q \Rightarrow (q = [])) \\
\quad | \text{HSChannel} - \Rightarrow \text{True}) \ g p
\end{center}

\texttt{exprArith} g p (ExprFull (ChanRef (VarRef gl name None))) =
\begin{center}
\texttt{trivCond} (withChannel' gl name None ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} \ cap - q \Rightarrow \text{integer-of-nat} (\text{length} q) \geq \text{cap} \\
\quad | \text{HSChannel} - \Rightarrow \text{False}) \ g p
\end{center}

\texttt{exprArith} g p (ExprFull (ChanRef (VarRef gl name (Some idx))) =
\begin{center}
\texttt{trivCond} (withChannel' gl name (Some (exprArith g p idx)) ( \\
\lambda \ c. \ \text{case} \ c \ \text{of} \\
\quad \text{Channel} \ cap - q \Rightarrow \text{integer-of-nat} (\text{length} q) \geq \text{cap} \\
\quad | \text{HSChannel} - \Rightarrow \text{False}) \ g p
\end{center}

\texttt{exprArith} g p (ExprVarRef (VarRef gl name None)) =
\begin{center}
\texttt{withVar'} gl name None id g p
\end{center}

\texttt{exprArith} g p (ExprVarRef (VarRef gl name (Some idx)) =
\begin{center}
\texttt{withVar'} gl name (Some (exprArith g p idx)) \ id g p
\end{center}
exprArith g p (ExprUnOp UnOpMinus expr) = 0 - exprArith g p expr
exprArith g p (ExprUnOp UnOpNeg expr) = ((exprArith g p expr) + 1) mod 2

exprArith g p (ExprBinOp BinOpAdd lexpr rexpr) =
(exprArith g p lexpr) + (exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpSub lexpr rexpr) =
(exprArith g p lexpr) - (exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpMul lexpr rexpr) =
(exprArith g p lexpr) * (exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpDiv lexpr rexpr) =
(exprArith g p lexpr) div (exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpMod lexpr rexpr) =
(exprArith g p lexpr) mod (exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpGr lexpr rexpr) =
trivCond (exprArith g p lexpr > exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpLe lexpr rexpr) =
trivCond (exprArith g p lexpr < exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpGEq lexpr rexpr) =
trivCond (exprArith g p lexpr ≥ exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpLEq lexpr rexpr) =
trivCond (exprArith g p lexpr ≤ exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpEq lexpr rexpr) =
trivCond (exprArith g p lexpr = exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpNEq lexpr rexpr) =
trivCond (exprArith g p lexpr ≠ exprArith g p rexpr)

exprArith g p (ExprBinOp BinOpAnd lexpr rexpr) =
trivCond (exprArith g p lexpr ≠ 0 ∧ exprArith g p rexpr ≠ 0)

exprArith g p (ExprBinOp BinOpOr lexpr rexpr) =
trivCond (exprArith g p lexpr ≠ 0 ∨ exprArith g p rexpr ≠ 0)

exprArith g p (ExprCond cexpr texpr fexpr) =
(if exprArith g p cexpr ≠ 0 then exprArith g p texpr
else exprArith g p fexpr)

exprArith g p (ExprPoll (ChanRef (VarRef gl name None)) rs srt) =
trivCond (withChannel' gl name None (}
\( \lambda \cdot c. \) pollCheck \( g \ p \ c \ \text{rs} \ \text{srt} \) \( g \ p \)

| exprArith \( g \ p \) (ExprPoll (ChanRef (VarRef gl name (Some idx))) \( \text{rs} \ \text{srt} \)) =
  trivCond (withChannel' gl name (Some (exprArith \( g \ p \) \( \text{idz} \)))
  (\( \lambda \cdot c. \) pollCheck \( g \ p \ c \ \text{rs} \ \text{srt} \) \( g \ p \))

| pollCheck \( g \ p \) InvChannel ' - - =
  abort "Channel already closed / invalid." \( \lambda \cdot \) False

| pollCheck \( g \ p \) (HSChannel -) - - = False

| pollCheck \( g \ p \) (Channel - - \( q \) \( \text{rs} \ \text{srt} \)) =
  if \( q = [] \) then False
  else if \( \neg \) srt then recvArgsCheck \( g \ p \ \text{rs} \) \( \text{hd} \ \text{q} \)
  else List.find (recvArgsCheck \( g \ p \ \text{rs} \)) \( q \neq \) None

| recvArgsCheck - - [] [] = True
| recvArgsCheck - - - [] =
  abort "Length mismatch on receiving." \( \lambda \cdot \) False
| recvArgsCheck - - [] - =
  abort "Length mismatch on receiving." \( \lambda \cdot \) False

| recvArgsCheck \( g \ p \) (r\#rs) (v\#vs) = ((
  case r of
    | RecvArgConst \( c \) \Rightarrow \( c = \) v
    | RecvArgMConst \( c \) \Rightarrow \( c = \) v
    | RecvArgVar \( var \) \Rightarrow \text{True}
    | RecvArgEval \( e \) \Rightarrow \text{exprArith} \( g \ p \ e = v \) \( \land \) recvArgsCheck \( g \ p \ \text{rs} \) \( \text{vs} \))

\text{getVar}' etc. do operate on name, index, \ldots directly. Lift them to use \textit{VarRef} instead.

\textbf{fun liftVar where}
\text{liftVar} \( f \) (VarRef gl v idx) \( \arg g p = \)
\( f gl v \) (map-option (exprArith \( g \ p \) \( \text{idz} \)) \( \arg g p \)

\textbf{definition getVar v = liftVar (\( \lambda \) gl v idx arg getVar' gl v idx) v ()}

\textbf{definition setVar = liftVar setVar'}

\textbf{definition withVar = liftVar withVar'}

\textbf{primrec withChannel}
\text{where} withChannel (ChanRef \( v \)) = \text{liftVar withChannel'} \( v \)

\textbf{lemma setVar-gState-progress-rel:}
\text{assumes gState-inv prog g}
\text{shows (g, \text{fst (setVar v val g p))} \in g\text{State-progress-rel prog}}
\text{(proof)}

\textbf{lemmas setVar-gState-inv =}
\text{setVar-gState-progress-rel[THEN gState-progress-rel-gState-invI2]}

\textbf{lemma setVar-pState-inv:}
\text{assumes pState-inv prog p}

44
shows \( p\text{State-inv prog (snd (setVar v val g p))} \)

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

**Lemma** \( \text{setVar-cl-inv}: \)
- **Assumes** \( \text{cl-inv (g,p)} \)
- **Shows** \( \text{cl-inv (setVar v val g p)} \)

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

### 5.4 Variable declaration

**Lemma** \( \text{channel-inv-code [code]}: \)
- **channel-inv** \( \text{(Channel cap ts q)} \)
- \( \text{\langle cap \leq max-array-size} \)
- \( \text{\& 0 \leq cap} \)
- \( \text{\& for-all \text{varType-inv} ts} \)
- \( \text{\& length ts \leq max-array-size} \)
- \( \text{\& length q \leq max-array-size} \)
- \( \text{\& for-all (\lambda x. \text{length x = length ts}} \)
- \( \text{\& for-all (\lambda y. y \geq min-var-value} \)
- \( \text{\& y \leq max-var-value) x)} q \)
- **channel-inv** \( \text{(HSChannel ts)} \)
- \( \text{\langle for-all \text{varType-inv} ts \& length ts \leq max-array-size} \)

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

**Primrec** \( \text{toVariable} \)
- \( \text{\langle \text{a gState-scheme} \Rightarrow pState \Rightarrow \text{varDecl} \Rightarrow \text{String.literal * variable * channels} \rangle} \)
- **Where**
  - \( \text{toVariable g p (VarDeclNum lb hb name siz init)} = ( \)
    - \( \text{let} \text{type} = \text{VTBoanded lb hb in} \)
    - \( \text{if} \neg \text{varType-inv type then abortv "Invalid var def (varType-inv failed): "} \)
    - \( \text{name} \)
    - \( \text{\langle \lambda-. \text{name, Var VTChan 0, []} \rangle} \)
  - \( \text{else} \)
    - \( \text{let} \)
    - \( \text{init} = \text{checkVarValue type \text{(case init of}} \)
    - \( \text{\text{None} \Rightarrow 0} \)
    - \( \text{| Some e \Rightarrow \text{exprArith g p e}}; \)
    - \( \text{v = (case siz of}} \)
    - \( \text{\text{None} \Rightarrow \text{Var type init}} \)
    - \( \text{| Some s \Rightarrow if \text{nat-of-integer s \leq max-array-size}} \)
    - \( \text{\text{then VArray type (nat-of-integer s)}} \)
    - \( \text{\text{\text{(IArray.tabulate (s, \lambda-. init))}} \}
    - \( \text{else abortv "Invalid var def (array too large): " \text{name}} \)
    - \( \text{\langle \lambda-. \text{Var VTChan 0) \rangle} \)
    - \( \text{in} \)
    - \( \text{(name, v, [])) \}
  - \( \text{| toVariable g p (VarDeclChan name siz types)} = ( \)
    - \( \text{let} \)

\( 45 \)
size = (case siz of None ⇒ 1 | Some s ⇒ nat-of-integer s);
chans = (case types of
  None ⇒ []
  | Some (cap, tys) ⇒
    let C = (if cap = 0 then HSChannel tys
    else Channel cap tys []) in
    if ¬ channel-inv C
    then abortv "Invalid var def (channel-inv failed):"
       name (λ- [])
    else replicate size C);
cidx = (case types of
  None ⇒ 0
  | Some - ⇒ integer-of-nat (length (channels g)));
v = (case siz of
  None ⇒ Var VTChan cidx
  | Some s ⇒ if nat-of-integer s ≤ max-array-size
    then VArray VTChan (nat-of-integer s)
       (IArray.tabulate (s,
        λi. if cidx = 0 then 0
         else i + cidx))
    else abortv "Invalid var def (array too large):"
       name (λ- Var VTChan 0))
in (name, v, chans)

lemma toVariable-variable-inv:
  assumes gState-inv prog g
  shows variable-inv (fst (snd (toVariable g p v)))))
⟨proof⟩
  including integer.lifting
⟨proof⟩

lemma toVariable-channels-inv:
  shows ∀ x ∈ set (snd (snd (toVariable g p v)))). channel-inv x
⟨proof⟩

lemma toVariable-channels-inv′:
  shows toVariable g p v = (a,b,c) ⇒ ∀ x ∈ set c. channel-inv x
⟨proof⟩

lemma toVariable-variable-inv′:
  shows gState-inv prog g ⇒ toVariable g p v = (a,b,c) ⇒ variable-inv b
⟨proof⟩

definition mkChannels
: 'a gState-scheme ⇒ pState ⇒ channels ⇒ 'a gState-scheme * pState
where
  mkChannels g p cs = (if cs = [] then (g,p) else
let l = length (channels g) in
if l + length cs > max-channels
then abort "Too much channels" (\-. (g,p))
else let
cs_p = map integer-of-nat [l..<l + length cs];
g' = g\[(channels := channels g @ cs)\];
p' = p\[(pState.channels := pState.channels p @ cs_p)\]
in
(g', p')

**lemma mkChannels-gState-progress-rel:**
gState-inv prog g
⇒ set cs ⊆ Collect channel-inv
⇒ (g, fst (mkChannels g p cs)) ∈ gState-progress-rel prog
⟨proof⟩

**lemmas mkChannels-gState-inv = mkChannels-gState-progress-rel[THEN gState-progress-rel-gState-invI2]**

**lemma mkChannels-pState-inv:**
pState-inv prog p
⇒ cl-inv (g,p)
⇒ pState-inv prog (snd (mkChannels g p cs))
⟨proof⟩

**including integer.lifting**
⟨proof⟩

**lemma mkChannels-cl-inv:**
cl-inv (g,p) ⇒ cl-inv (mkChannels g p cs)
⟨proof⟩

**definition mkVarChannel**
:: varDecl
⇒ ((var-dict ⇒ var-dict) ⇒ 'a gState-scheme * pState
⇒ 'a gState-scheme ⇒ pState
⇒ 'a gState-scheme * pState
where
mkVarChannel v upd g p = (let
(k,v,cs) = toVariable g p v;
(g',p') = upd (lm.update k v) (g,p)
in
mkChannels g' p' cs)

**lemma mkVarChannel-gState-inv:**
assumes gState-inv prog g
and \(\forall k v' cs. toVariable g p v = (k,v',cs)\)
⇒ gState-inv prog (fst (upd (lm.update k v') (g,p)))
shows \( gState-inv \ \text{prog} (\text{fst} (\text{mkVarChannel } v \ \text{upd} \ g \ p)) \)

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

\textbf{lemma} \( \text{mkVarChannel-gState-progress-rel} \):
\textbf{assumes} \( gState-inv \ \text{prog} \ g \)
\textbf{and} \( \forall k \ v' \ cs. \ \text{toVariable} \ g \ p \ v = (k, v', cs) \)
\( \implies (g, \ \text{fst} (\text{upd} (\text{lm.update} k \ v') \ (g, p))) \in gState-progress-rel \ \text{prog} \)
\textbf{shows} \( (g, \ \text{fst} (\text{mkVarChannel } v \ \text{upd} \ g \ p)) \in gState-progress-rel \ \text{prog} \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{mkVarChannel-pState-inv} \):
\textbf{assumes} \( pState-inv \ \text{prog} \ p \)
\textbf{and} \( \text{cl-inv} \ (g, p) \)
\( \forall k \ v' \ cs. \ \text{toVariable} \ g \ p \ v = (k, v', cs) \)
\( \implies \text{cl-inv} (\text{upd} (\text{lm.update} k \ v') \ (g, p)) \)
\textbf{and} \( \forall k \ v' \ cs. \ \text{toVariable} \ g \ p \ v = (k, v', cs) \)
\( \implies pState-inv \ \text{prog} (\text{snd} (\text{upd} (\text{lm.update} k \ v') \ (g, p))) \)
\textbf{shows} \( pState-inv \ \text{prog} (\text{snd} (\text{mkVarChannel } v \ \text{upd} \ g \ p)) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{mkVarChannel-cl-inv} \):
\textbf{assumes} \( \text{cl-inv} \ (g, p) \)
\textbf{and} \( \forall k \ v' \ cs. \ \text{toVariable} \ g \ p \ v = (k, v', cs) \)
\( \implies \text{cl-inv} (\text{upd} (\text{lm.update} k \ v') \ (g, p)) \)
\textbf{shows} \( \text{cl-inv} (\text{mkVarChannel } v \ \text{upd} \ g \ p) \)
\( \langle \text{proof} \rangle \)

\textbf{definition} \( \text{mkVarChannelProc} \):
\( :: \ \text{procVarDecl} \Rightarrow 'a \ \text{gState-scheme} \Rightarrow \text{pState} \Rightarrow 'a \ \text{gState-scheme} \ast \ \text{pState} \)
\textbf{where} \n
\( \text{mkVarChannelProc} \ v \ g \ p = ( \)
\textbf{let} \n
\( v' = \text{case } v \ \text{of} \)
\( \text{ProcVarDeclNum} \ lb \ hb \ name \ siz \ init \Rightarrow \text{VarDeclNum} \ lb \ hb \ name \ siz \ init \)
| \text{ProcVarDeclChan} \ name \ siz \Rightarrow \text{VarDeclChan} \ name \ siz \ None; \)
\( \ (k, v, cs) = \text{toVariable} \ g \ p \ v' \)
\textbf{in} \n
\( \text{mkVarChannel} \ v' \ (\text{apsnd} \circ \text{pState.vars-update}) \ g \ p) \)

\textbf{lemma} \( \text{mkVarChannelProc-gState-progress-rel} \):
\textbf{assumes} \( gState-inv \ \text{prog} \ g \)
\textbf{shows} \( (g, \ \text{fst} (\text{mkVarChannelProc} \ v \ g \ p)) \in gState-progress-rel \ \text{prog} \)
\( \langle \text{proof} \rangle \)

\textbf{lemmas} \( \text{mkVarChannelProc-gState-inv} = \)
\( \text{mkVarChannelProc-gState-progress-rel} [\text{THEN} gState-progress-rel-gState-invI2] \)
lemma toVariable-name:
  toVariable g p (VarDeclNum lb hb name sz init) = (x,a,b) \rightarrow x = name
  toVariable g p (VarDeclChan name sz t) = (x, a, b) \rightarrow x = name
⟨proof⟩
delect toVariable.simps[simp del]

lemma statesDecls-process-names:
  assumes v \in statesDecls (states prog !! (pState.idx p))
  shows procVarDeclName v \in process-names (states prog !! (pState.idx p))
⟨proof⟩

lemma mkVarChannelProc-pState-inv:
  assumes pState-inv prog p
  and gState-inv prog g
  and cl-inv (g,p)
  and decl: v \in statesDecls (states prog !! (pState.idx p))
  shows pState-inv prog (snd (mkVarChannelProc v g p))
⟨proof⟩

lemma mkVarChannelProc-cl-inv:
  assumes cl-inv (g,p)
  shows cl-inv (mkVarChannelProc v g p)
⟨proof⟩

5.5 Folding

Fold over lists (and lists of lists) of step/stmnt. The folding functions are doing a bit more than that, e.g. ensuring the offset into the program array is correct.

definition step-fold' where
  step-fold' g steps (lbs :: labels) pri pos
  (nxt :: edgeIndex) (onxt :: edgeIndex option) iB =
  foldr (λstep (pos, nxt, lbs, es).
  let (e, enxt, lbs) = g step (lbs, pri, pos, nxt, onxt, iB)
  in (pos + length e, enxt, lbs, es@e)
) steps (pos, nxt, lbs, [])

definition step-fold where
  step-fold g steps lbs pri pos nxt onxt iB = (          
  let (s,nxt,lbs,s) = step-fold' g steps lbs pri pos nxt onxt iB
  in (s,nxt,lbs))

lemma step-fold'-cong:
  assumes lbs = lbs'
  and pri = pri'
  and pos = pos'
  and steps = steps'
\begin{verbatim}
and \( \text{nxt} = \text{nxt}' \)
and \( \text{onxt} = \text{onxt}' \)
and \( \text{iB} = \text{iB}' \)
and \( \bigwedge x d, x \in \text{set steps} \Rightarrow g x d = g' x d \)
shows \( \text{step-fold'} g \text{ steps } \text{lbls } \text{pri } \text{pos } \text{onxt } \text{iB} = \text{step-fold'} g' \text{ steps }' \text{lbls }' \text{pri}' \text{ pos}' \text{ onxt}' \text{iB}' \)
\end{verbatim}

\textbf{lemma} \text{step-fold-cong}[fundef-cong]:
\begin{verbatim}
assumes \( \text{lbls} = \text{lbls}' \)
and \( \text{pri} = \text{pri}' \)
and \( \text{pos} = \text{pos}' \)
and \( \text{steps} = \text{steps}' \)
and \( \text{nxt} = \text{nxt}' \)
and \( \text{onxt} = \text{onxt}' \)
and \( \text{iB} = \text{iB}' \)
and \( \bigwedge x d, x \in \text{set steps} \Rightarrow g x d = g' x d \)
shows \( \text{step-fold} g \text{ steps } \text{lbls } \text{pri } \text{pos } \text{onxt } \text{iB} = \text{step-fold} g' \text{ steps }' \text{lbls }' \text{pri}' \text{ pos}' \text{ onxt}' \text{iB}' \)
\end{verbatim}

\textbf{fun} \text{step-foldL-step where}
\begin{verbatim}
\text{step-foldL-step} - - - [] (\text{pos}, \text{nxt}, \text{lbls}, \text{es}, \text{is}) = (\text{pos}, \text{nxt}, \text{lbls}, \text{es}, \text{is})
| \text{step-foldL-step} g \text{ pri } \text{onxt} (s#\text{steps}) (\text{pos}, \text{nxt}, \text{lbls}, \text{es}, \text{is}) = (\text{let} (\text{pos}', \text{nxt}', \text{lbls}', \text{ss}') = \text{step-fold'} g \text{ steps } \text{lbls } \text{pri } \text{pos } \text{onxt } \text{False in}
| \text{let} (s', \text{nxt}'', \text{lbls}'') = g s (\text{lbs}', \text{pri}.\text{pos}', \text{nxt}', \text{onxt}, \text{True}) \text{ in}
| \text{let} \text{rs} = \text{butlast s'}; s'' = \text{last s'} \text{ in}
| (\text{pos}' + \text{length rs}, \text{nxt}, \text{lbls}''', \text{es}@\text{ss}@\text{rs}, \text{s''#is}))
\end{verbatim}

\textbf{definition} \text{step-foldL where}
\begin{verbatim}
\text{step-foldL} g \text{ stepss } \text{lbls } \text{pri } \text{pos } \text{onxt } \text{iB} = \text{foldr} (\text{step-foldL-step} g \text{ pri } \text{onxt}) \text{ stepss } (\text{pos},\text{nxt},\text{lbs},[],[])
\end{verbatim}

\textbf{lemma} \text{step-foldL-step-cong}:
\begin{verbatim}
assumes \( \text{pri} = \text{pri}' \)
and \( \text{onxt} = \text{onxt}' \)
and \( s = s' \)
and \( d = d' \)
and \( \bigwedge x d, x \in \text{set s} \Rightarrow g x d = g' x d \)
shows \( \text{step-foldL-step} g \text{ pri } \text{onxt } s d = \text{step-foldL-step} g' \text{ pri}' \text{ onxt}' s' d' \)
\end{verbatim}

\textbf{lemma} \text{step-foldL-cong}[fundef-cong]:
\begin{verbatim}
assumes \( \text{lbls} = \text{lbls}' \)
and \( \text{pri} = \text{pri}' \)
and \( \text{pos} = \text{pos}' \)
and \( \text{stepss} = \text{stepss}' \)
and \( \text{nxt} = \text{nxt}' \)
and \( \text{onxt} = \text{onxt}' \)
\end{verbatim}
\( \land x' \ d, \ x \in \text{set steps} \implies x' \in \text{set} x \implies g \ x' \ d = g' \ x' \ d \)

**shows** \( \text{step-foldL} \ g \ \text{steps} \ \text{lbs} \ \text{pri} \ \text{pos} \ \text{nxt} \ \text{onxt} = \text{step-foldL} \ g' \ \text{steps}' \ \text{lbs}' \ \text{pri}' \ \text{pos}' \ \text{nxt}' \ \text{onxt}' \)

\[\text{(proof)}\]

### 5.6 Starting processes

**definition** \( \text{modProcArg} \) :: \((\text{procArg} \ast \text{integer}) \Rightarrow \text{String.literal} \ast \text{variable}\)

**where**

\( \text{modProcArg} \ x = (\text{case} \ x \ \text{of} \ (\text{ProcArg} \ ty \ \text{name}, \ \text{val}) \ \Rightarrow \ \text{if} \ \text{varType-inv} \ \text{ty} \ \text{then} \ \text{let} \ \text{init} = \ \text{checkVarValue} \ \text{ty} \ \text{val} \ \text{in} \ (\text{name}, \ \text{Var} \ \text{ty} \ \text{init}) \ \text{else} \ \text{abortv} \text{ "Invalid proc arg (varType-inv failed)"} \ \text{name} (\lambda-. \ (\text{name}, \ \text{Var} \ \text{VTChan} \ 0))) \)

**definition** \( \text{emptyProc} :: \text{pState} \)

— The empty process.

**where**

\( \text{emptyProc} = (| \text{pid} = 0, \ \text{vars} = \text{lm.empty} (), \ \text{pc} = 0, \ \text{channels} = [], \ \text{idx} = 0 |) \)

**lemma** \( \text{vardict-inv-empty} : \text{vardict-inv ss proc (lm.empty())} \)

\[\text{(proof)}\]

**lemma** \( \text{emptyProc-cl-inv[simp]} : \text{cl-inv (g, emptyProc)} \)

\[\text{(proof)}\]

**lemma** \( \text{emptyProc-pState-inv} : \text{assumes program-inv prog} \ \text{shows pState-inv prog emptyProc} \)

\[\text{(proof)}\]

**fun** \( \text{mkProc} :: 'a \text{gState-scheme} \Rightarrow \text{pState} \)

\( \Rightarrow \text{String.literal} \Rightarrow \text{expr list} \Rightarrow \text{process} \Rightarrow \text{nat} \)

\( \Rightarrow 'a \text{gState-scheme} \ast \text{pState} \)

**where**

\( \text{mkProc} \ g \ p \ \text{name args (sidx, start, argDecls, decls) pidN = (let start = case start of} \)

\( \quad \text{Index x} \Rightarrow x \)

\( \quad | - \Rightarrow \text{abortv "Process start is not index: " name (\lambda-. 0)} \)

\( \text{in} \)

\( (* \ \text{sanity check} *) \)

\( \text{if} \ \text{length args} \neq \text{length argDecls} \)

\( \text{then} \ \text{abortv "Signature mismatch: " name (\lambda-. (g, emptyProc))} \)
else
let
(* evaluate args (in the context of the calling process) *)
eArgs = map (exprArith g p) args;

(* replace the init part of argDecls *)
argVars = map modProcArg (zip argDecls eArgs);

(* add -pid to vars *)
pidI = integer-of-nat pidN;
argVars = (STR "-pid", Var (VTBounded 0 pidI) pidI)#argVars;
argVars = lm.to-map argVars;

(* our new process *)
p = (| pid = pidN, vars = argVars, pc = start, channels = [], idx = sidx |)
in
(* apply the declarations *)
foldl (\(g, p\) d. mkVarChannel d (apsnd \circ pState.vars-update) g p)
\((g, p)\)
decls)

lemma mkProc-gState-progress-rel:
assumes gState-inv prog g
shows (g, fst (mkProc g p name args (processes prog !! sidx) pidN))) ∈
gState-progress-rel prog
⟨proof⟩
lemmas mkProc-gState-inv = mkProc-gState-progress-rel[THEN gState-progress-rel-gState-invI2]

lemma mkProc-pState-inv:
assumes program-inv prog
and gState-inv prog g
and pidN ≤ max-procs and pidN > 0
and sidx < IArray.length (processes prog)
and fst (processes prog !! sidx) = sidx
shows pState-inv prog (snd (mkProc g p name args (processes prog !! sidx) pidN))
⟨proof⟩
including integer.lifting
⟨proof⟩

lemma mkProc-cl-inv:
assumes cl-inv (g,p)
shows cl-inv (mkProc g p name args (processes prog !! sidx) pidN)
⟨proof⟩
define runProc
:: String.literal ⇒ expr list ⇒ program
⇒ 'a gState-scheme ⇒ pState
⇒ 'a gState-scheme * pState

where
runProc name args prog g p = (  
  if length (procs g) ≥ max-procs  
  then abort "Too many processes" (λ.- (g,p)) 
  else let pid = length (procs g) + 1 in  
    case lm.lookup name (proc-data prog) of  
      None ⇒ abortv "No such process: " name  
        (λ.- (g,p))  
      Some proc-idx ⇒  
        let  
          (g′, proc) = mkProc g p name args (processes prog !! proc-idx) pid  
        in (g′(procs := procs g @ [proc]), p))

lemma runProc-gState-progress-rel:
  assumes program-inv prog 
  and gState-inv prog g 
  and pState-inv prog p 
  and cl-inv (g, p) 
  shows (g, fst (runProc name args prog g p)) ∈ gState-progress-rel prog
⟨proof⟩

lemmas runProc-gState-inv  
  runProc-gState-progress-rel[THEN gState-progress-rel-gState-invI2]

lemma runProc-pState-id:
  snd (runProc name args prog g p) = p
⟨proof⟩

lemma runProc-pState-inv:
  assumes pState-inv prog p 
  shows pState-inv prog (snd (runProc name args prog g p))
⟨proof⟩

lemma runProc-cl-inv:
  assumes program-inv prog 
  and gState-inv prog g 
  and pState-inv prog p 
  and cl-inv (g, p) 
  shows cl-inv (runProc name args prog g p)
⟨proof⟩

5.7 AST to edges

type-synonym ast = AST.module list

In this section, the AST is translated into the transition system.

Handling atomic blocks is non-trivial. Therefore, we do this in an extra pass:
lp and hp are the positions of the start and the end of the atomic block. Every edge pointing into this range is therefore marked as Atomic. If they are pointing somewhere else, they are set to InAtomic, meaning: they start in an atomic block, but leave it afterwards.

**definition** atomize :: nat ⇒ nat ⇒ edge list ⇒ edge list where
atomize lp hp es = fold (λe es.
let e’ = case target e of
      LabelJump - None ⇒
      (*
        Labels are checked again later on, when they
        are going to be resolved. Hence it is safe to say
        'atomic' here, especially as the later algorithm
        relies on targets in atomic blocks to be marked as such.
      *)
    e\(\)\ atomic := InAtomic \)
  | LabelJump - (Some via) ⇒
    if lp ≤ via ∧ hp ≥ via then e\(\)\ atomic := Atomic \)
    else e\(\)\ atomic := InAtomic \)
  | Index p’ ⇒
    if lp ≤ p’ ∧ hp ≥ p’ then e\(\)\ atomic := Atomic \)
    else e\(\)\ atomic := InAtomic \)
in e’\#es) es []

fun skip — No-op edge
where
skip (lbs, pri, pos, nxt, -) =
  ([(cond = EExpr (ExprConst 1),
     effect = EEId, target = nxt, prio = pri,
     atomic = NonAtomic)], Index pos, lbs)

The AST is walked backwards. This allows to know the next state directly.
Parameters used:

lbs Map of Labels
pri Current priority
pos Current position in the array
nxt Next state
onxt Previous 'next state' (where to jump after a 'do')
inBlock Needed for certain constructs to calculate the layout of the array

fun stepToState
:: step
⇒ (labels * integer * nat * edgeIndex * edgeIndex option * bool)
⇒ edge list list * edgeIndex * labels

54
and stmtToState
:: stmt
⇒ (labels * integer * nat * edgeIndex * edgeIndex option * bool)
⇒ edge list list * edgeIndex * labels

where

stepToState (StepStmt s None) data = stmtToState s data
| stepToState (StepStmt s (Some u)) (bls, pri, pos, nxt, onxt, -) = (let
  (* the ‘unless’ part *)
  (ues,-,bls') = stmtToState u (bls, pri, pos, nxt, onxt, True);
  u = last ues; ues = butlast ues;
  pos' = pos + length ues;

  (* find minimal current priority *)
  pri = min-prio u pri;

  (* the guarded part —
   priority is decreased, because there is now a new unless part with
   higher prio *)
  (ses,spos,lbls'') = stmtToState s (bls', pri - 1, pos', nxt, onxt, False);

  (* add an edge to the unless part for each generated state *)
  ses = map (List.append u) ses
in
  (ues@ses,spos,lbls''))

| stepToState (StepDecl decls) (bls, pri, pos, nxt, onxt, -) = (let edgeF = λ d (bls,pri,pos,nxt,-).
  (\[(\[cond = ECTrue, effect = EEDecl d, target = nxt,
    prio = pri, atomic = NonAtomic]\]), Index pos, bls)
  in
  step-fold edgeF decls bls pri pos nxt onxt False)

| stepToState StepSkip (bls,-,-,nxt,-) = ([],nxt,bls)

| stmtToState (StmtAtomic steps) (bls, pri, pos, nxt, onxt, inBlock) = (let es,pos',bls') = step-fold stepToState steps bls pri pos nxt onxt inBlock in
  let es' = map (atomize pos (pos + length es)) es in
  (es', pos', bls'))

| stmtToState (StmtLabeled l s) (bls, pri, pos, d) = (let
  (es, pos', bls) = stmtToState s (bls, pri, pos, d);

  (* We don’t resolve goto–chains. If the labeled stmt returns only a jump,
   use this goto state. *)
  lpos = case pos' of Index p ⇒ p | _ ⇒ pos;
  bls' = add-label l bls lpos
in

55
(es, pos’, lbls’))

| stmtToState (StmntDo stepss) (lbs, pri, pos, nxt, onxt, inBlock) = (let
  (*
   construct the different branches
   ’nxt’ in those branches points current pos (it is a loop after all)
   ’onxt’ then is the current ’nxt’ (needed for break, f.ex.)
   *)
  (lbs, es, is) = step-foldL stepToState stepss lbs pri
                 (pos + 1) (Index pos) (Some nxt);

  (* put the branch starting points (’is’) into the array *)
  es’ = concat is # es

  in
    if inBlock then
      (* inside another DO or IF or UNLESS
       -> append branches again, so they can be consumed *)
      (es’ @ [concat is], Index pos, lbs)
    else
      (es’, Index pos, lbs)
  )

| stmtToState (StmntIf stepss) (lbs, pri, pos, nxt, onxt, -) = (let
  (pos’, lbs, es, is) = step-foldL stepToState stepss lbs pri pos nxt onxt
  in (es @ [concat is], Index pos, lbs))

| stmtToState (StmntSeq steps) (lbs, pri, pos, nxt, onxt, inBlock) =
  step-fold stepToState steps lbs pri pos nxt onxt inBlock

| stmtToState (StmntAssign v e) (lbs, pri, pos, nxt, -) =
  ([[[cond = ECTrue, effect = EEAssign v e, target = nxt, prio = pri,
     atomic = NonAtomic]]], Index pos, lbs)

| stmtToState (StmntAssert e) (lbs, pri, pos, nxt, -) =
  ([[[cond = ECTrue, effect = EEAssert e, target = nxt, prio = pri,
     atomic = NonAtomic]]], Index pos, lbs)

| stmtToState (StmntCond e) (lbs, pri, pos, nxt, -) =
  ([[[cond = ECExpr e, effect = EEId, target = nxt, prio = pri,
     atomic = NonAtomic]]], Index pos, lbs)

| stmtToState StmntElse (lbs, pri, pos, nxt, -) =
  ([[[cond = ECElse, effect = EEId, target = nxt, prio = pri,
     atomic = NonAtomic]]], Index pos, lbs)

| stmtToState StmntBreak (lbs, pri, -,-,Some onxt,-) =
  ([[[cond = ECTrue, effect = EEGoto, target = onxt, prio = pri,
atomic = NonAtomic [], onxt, lbls)
| stmtToState StmntBreak (\x\n.\n.None,-) =
  abort "Misplaced break" (\_. ([], Index 0, lm.empty()))

| stmtToState (StmntRun n args) (l lbls, pri, pos, nxt, onxt, -) =
  ([|\{ cond = ECRun n, effect = EERun n args, target = nxt, prio = pri,
      atomic = NonAtomic [] |], Index pos, lbls)

| stmtToState (StmntGoTo l) (l lbls, pri, pos, -, -) =
  ([|\{ cond = ECTrue, effect = EEGoto, target = LabelJump l None, prio = pri,
      atomic = NonAtomic [] |], LabelJump l (Some pos), lbls)

| stmtToState (StmntSend v e srt) (l lbls, pri, pos, nxt, -) =
  ([|\{ cond = ECSend v, effect = EESend v e srt, target = nxt, prio = pri,
      atomic = NonAtomic [] |], Index pos, lbls)

| stmtToState (StmntRecv v r srt rem) (l lbls, pri, pos, nxt, -) =
  ([|\{ cond = ECRecv v r srt, effect = EERecv v r srt rem, target = nxt, prio =
      pri, atomic = NonAtomic [] |], Index pos, lbls)

| stmtToState StmntSkip d = skip d

5.7.1 Setup

definition endState :: edge list where
  — An extra state added to each process marking its end.
  endState = [|\{ cond = ECFalse, effect = EEEnd, target = Index 0, prio = 0,
      atomic = NonAtomic [] |]

definition resolveLabel :: String.literal ⇒ labels ⇒ nat where
  resolveLabel l lbls = (case lm.lookup l lbls of
    None ⇒ abortv "Unresolved label: " l (\_. 0)
    Some pos ⇒ pos)

primrec resolveLabels :: edge list list ⇒ labels ⇒ edge list ⇒ edge list where
  resolveLabels - - [] = []
| resolveLabels edges lbls (e#es) = (let check-atomic = \pos. fold (\e a. a ∧ inAtomic e) (edges ! pos) True in
case target e of
  Index - ⇒ e
| LabelJump l None ⇒
  let pos = resolveLabel l lbls in
  e[target := Index pos, atomic := if inAtomic e then
    if check-atomic pos then Atomic
    else InAtomic
    else NonAtomic] )
LabelJump l (Some via) ⇒
let pos = resolveLabel l lbls in
e[target] := Index pos,
(* NB: isAtomic instead of inAtomic, of atomize(*) *)
atomic := if isAtomic e then
  if check-atomic pos ∧ check-atomic via then Atomic
  else InAtomic
else atomic e ⌜
⟩≡ \ (resolveLabels edges lbls es)
definition calculatePrios :: edge list list ⇒ (integer ∗ edge list) list where
calculatePrios ess = map (λes. (min-prio es 0, es)) ess
definition toStates :: step list ⇒ states ∗ edgeIndex ∗ labels where
toStates steps = (let
  (states, pos, lbls) = step-fold stepToState steps (lm.empty())
  0 1 (Index 0) None False;
  pos = (case pos of
      Index - ⇒ pos
      | LabelJump l - ⇒ Index (resolveLabel l lbls));
  states = endState ≠ states;
  states = map (resolveLabels states lbls) states;
  states = calculatePrios states
in
  case pos of
      Index s ⇒
        if s < length states then (IArray states, pos, lbls)
        else abort "Start index out of bounds" (λ-.
            (IArray states, Index 0, lbls)))
      toStates steps = (ss,start,lbls)
definition toProcess :: nat ⇒ proc ⇒ states ∗ nat ∗ String.literal ∗ (labels ∗ process) where
toProcess sidx (ProcType act name args decls steps) = (let
  (states, start, lbls) = toStates steps;
  act = (case act of
      None ⇒ 0
      | Some None ⇒ 1
      | Some (Some x) ⇒ nat-of-integer x
  in
      (states, act, name, lbls, sidx, start, args, decls))
| toProcess sidx (Init decls steps) = (}
lemma toStates-inv:
  assumes toStates steps = (ss,start,lbls)
  shows ∃s. start = Index s ∧ s < IArray.length ss
      and IArray.length ss > 0
(proof)
let (states, start, lbls) = toStates steps in
(states, 1, STR "init", lbls, sidz, start, [], decls))

lemma toProcess-sidx:
toProcess sidx p = (ss,a,n,l,idx,r) ⇒ idx = sidx
⟨proof⟩

lemma toProcess-states-nonempty:
toProcess sidx p = (ss,a,n,l,idx,r) ⇒ IArray.length ss > 0
⟨proof⟩

lemma toProcess-start:
toProcess sidx p = (ss,a,n,l,idx,r)⇒ ∃s. start = Index s ∧ s < IArray.length ss
⟨proof⟩

lemma toProcess-startE:
assumes toProcess sidx p = (ss,a,n,l,idx,r) obtains s where start = Index s ∧ s < IArray.length ss
⟨proof⟩

The main construction function. Takes an AST and returns an initial state,
and the program (= transition system).

definition setUp :: ast ⇒ program × gState where
setUp ast = (
let
   (decls, proc, -) = preprocess ast;
   assertVar = Var (VTBounded 0 1) 0;

   pre-procs = map (split toProcess) (List.enumerate 1 procs);

   proces = IArray ((0, Index 0, [], []) # map (λ(-,-,-,-,p). p) pre-procs);
   labels = IArray (lm.empty() # map (λ(-,-,l,-) l) pre-procs);
   states = IArray (IArray [(0,[])]) # map (λ(s,). s) pre-procs);
   names = IArray (STR "invalid" # map (λ(-,-,n,-) n) pre-procs);

   proc-data = lm.to-map (map (λ(-,-n,-,idx,-) (n,idx)) pre-procs);

   prog = ([] processes = proces, labels = labels, states = states,
   proc-names = names, proc-data = proc-data [])
;

   g = ([] vars = lm.sng (STR "-assert-") assertVar,
       channels = [InvChannel], timeout = False, procs = [] []);

   g' = foldl (λg d.
                 fst (mkVarChannel d (apfst o gState.vars-update) g emptyProc)
               ) g decls;

   g" = foldl (λg (-,-a,name,-).
               foldl (λg name.

59
\[ \text{fst } \text{(runProc name [] prog g emptyProc)} \]
\[ \text{g } \text{(replicate a name)} \]
\[ \text{g' } \text{pre-procs} \]
\[ \text{in } \]
\[ (\text{prog, g'}) \]

**Lemma** `setUp-program-inv`:
\[ \text{program-inv } (\text{fst } (\text{setUp ast})) \]

**Proof**

**Lemma** `setUp-program-inv`:
\[ \text{assumes } \text{setUp ast }= (\text{prog, g}) \]
\[ \text{shows } \text{program-inv prog} \]

**Proof**

**Lemma** `setUp-gState-inv`:
\[ \text{assumes } \text{setUp ast }= (\text{prog, g}) \]
\[ \text{shows } \text{gState-inv prog g} \]

**Proof**

### 5.8 Semantic Engine

After constructing the transition system, we are missing the final part: The successor function on this system. We use SPIN-nomenclature and call it **semantic engine**.

**Definition** `assertVar` ≡ `VarRef True (STR "--assert--") None`

#### 5.8.1 Evaluation of Edges

**Function** `evalRecvArgs` :: `recvArg list ⇒ integer list ⇒ gState I ⇒ pState ⇒ gState I * pState`

**Where**

- `evalRecvArgs [] [] g l = (g,l)`
- `evalRecvArgs - [] g l = abort "Length mismatch on receiving." (λ- (g,l))`
- `evalRecvArgs [] - g l = abort "Length mismatch on receiving." (λ- (g,l))`
- `evalRecvArgs (r#rs) (v#vs) g l = (let (g,l) = case r of `RecvArgVar var ⇒ setVar var v g l
  | - ⇒ (g,l)
in evalRecvArgs rs vs g l)`

**Primrec** `evalCond` :: `edgeCond ⇒ gState I ⇒ pState ⇒ bool`

**Where**

- `evalCond ECTrue - - ⇔ True`
- `evalCond ECTrue - - ⇔ False`
evalCond (ECExpr e) g l ←→ exprArith g l e ≠ 0
evalCond (ECRun -) g l ←→ length (procs g) < 255
evalCond ECElse g l ←→ gStateΙ.else g
evalCond (ECSend v) g l ←→
  withChannel v (λ- c.
    case c of
      Channel cap - q ⇒ integer-of-nat (length q) < cap
    | HSChannel - ⇒ True) g l
  withChannel v (λi c.
    case c of
      HSChannel - ⇒ handshake g ≠ 0 ∧ recvArgsCheck g l rs (hsdata g)
    | - ⇒ pollCheck g l c rs srt) g l

fun evalHandshake
:: edgeCond ⇒ nat ⇒ gStateΙ ⇒ pState ⇒ bool
where
evalHandshake (ECRecv v - -) h g l
  ←→ h = 0
  ∨ withChannel v (λi c. case c of
    HSChannel - ⇒ i = h
  | Channel - - ⇒ False) g l
  withChannel v (λi c.
    case c of
      HSChannel - ⇒ handshake g ≠ 0 ∧ recvArgsCheck g l rs (hsdata g)
    | - ⇒ pollCheck g l c rs srt) g l

primrec evalEffect
:: edgeEffect ⇒ program ⇒ gStateΙ ⇒ pState ⇒ gStateΙ * pState
where
evalEffect EEEnd - g l = (g, l)
| evalEffect EEId - g l = (g, l)
| evalEffect EEGoto - g l = (g, l)
| evalEffect (EEAssign v e) - g l = setVar v (exprArith g l e) g l
| evalEffect (EEDecl d) - g l = mkVarChannelProc d g l
| evalEffect (EERun name args) prog g l = runProc name args prog g l
| evalEffect (EEAssert e) - g l =
  if exprArith g l e = 0
  then setVar assertVar 1 g l
  else (g, l))
| evalEffect (EESend v es srt) - g l = withChannel v (λi c.
  let
    ab = λ-. abort "Length mismatch on sending." (λ-. (g, l));
    es = map (exprArith g l) es
  in
  if ¬ for-all (λx. x ≥ min-var-value ∧ x ≤ max-var-value) es
  then abort "Invalid Channel" (λ-. (g, l))
  else
    case c of
      Channel cap - q ⇒
        if length ts ≠ length es ∨ ¬ (length q < max-array-size)
        then ab()
else let
  q′ = if ¬ srt then q@[es]
  else let
    q = map leq list q;
    q′ = insort (leq list es) q
    in map unleq q′;
    g = gState.channels-update (λcs.
      cs[i := Channel cap ts q′]) g
  in (g,l)
| HSCall ts ⇒
  if length ts ≠ length es then ab()
  else (g[[hsdata := es, handshake := i], l])
| InvChannel ⇒ abort "Trying to send on invalid channel" (λ- (g,l))
) g l
| evalEffect (EERecv v rs srt rem) - g l = withChannel v (λi c of
  Channel cap ts qs ⇒
  if qs = [] then abort "Recv from empty channel" (λ- (g,l))
  else let
    (q′, qs′) = if ¬ srt then (hd qs, tl qs)
      else apfst the (find-remove (recvArgsCheck g l rs) qs);
    (g,l) = evalRecvArgs rs q′ g l;
    g = if rem
      then gState.channels-update (λcs. cs[i := Channel cap ts qs′]) g
      else g
      (* messages are not removed -- so no need to update anything *)
    in (g,l)
| HSCall - ⇒
  let (g,l) = evalRecvArgs rs (hsdata g) g l in
  let g = g[[handshake := 0, hsdata := []]]
  in (g,l)
| InvChannel ⇒ abort "Receiving on invalid channel" (λ- (g,l))
) g l

lemma statesDecls-effect:
  assumes ef ∈ effect ' edgeSet ss
  and ef = EEDecl d
  shows d ∈ statesDecls ss
⟨proof⟩

lemma evalRecvArgs-pState-inv:
  assumes pState-inv prog p
  shows pState-inv prog (snd (evalRecvArgs rargs xs g p))
⟨proof⟩

lemma evalRecvArgs-pState-inv':
  assumes evalRecvArgs rargs xs g p = (g′, p′)
  and pState-inv prog p
62
shows $pState-inv \ prog \ p'$
(\textit{proof})

\textbf{lemma} \textit{evalRecvArugs-gState-progress-rel}:
\begin{itemize}
\item \textbf{assumes} $gState-inv \ prog \ g$
\item \textbf{shows} $(g, \ \text{fst} (\textit{evalRecvArugs} \ rargs \ xs \ g \ p)) \in gState-progress-rel \ prog$
\end{itemize}
(\textit{proof})

\textbf{lemmas} \textit{evalRecvArugs-gState-inv} = \textit{evalRecvArugs-gState-progress-rel}[\textit{THEN} \ gState-progress-rel-gState-invI2]

\textbf{lemma} \textit{evalRecvArugs-cl-inv}:
\begin{itemize}
\item \textbf{assumes} $cl-inv (g, p)$
\item \textbf{shows} $cl-inv (\textit{evalRecvArugs} \ rargs \ xs \ g \ p)$
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{evalEffect-pState-inv}:
\begin{itemize}
\item \textbf{assumes} $pState-inv \ prog \ p$
\item \textbf{and} $gState-inv \ prog \ g$
\item \textbf{and} $cl-inv (g, p)$
\item \textbf{and} $e \in \text{effect} \cdot \text{edgeSet} (\text{states} \ prog \ \text{!!} \ pState.\idx \ p)$
\item \textbf{shows} $pState-inv \ prog \ (\text{snd} (\textit{evalEffect} \ e \ prog \ g \ p))$
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{evalEffect-gState-progress-rel}:
\begin{itemize}
\item \textbf{assumes} $program-inv \ prog$
\item \textbf{and} $gState-inv \ prog \ g$
\item \textbf{and} $pState-inv \ prog \ p$
\item \textbf{and} $cl-inv (g, p)$
\item \textbf{shows} $(g, \ \text{fst} (\textit{evalEffect} \ e \ prog \ g \ p)) \in gState-progress-rel \ prog$
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{evalEffect-cl-inv}:
\begin{itemize}
\item \textbf{assumes} $cl-inv (g, p)$
\item \textbf{and} $program-inv \ prog$
\item \textbf{and} $gState-inv \ prog \ g$
\item \textbf{and} $pState-inv \ prog \ p$
\item \textbf{shows} $cl-inv (\textit{evalEffect} \ e \ prog \ g \ p)$
\end{itemize}
(\textit{proof})

\section{Executable edges}

To find a successor global state, we first need to find all those edges which are executable (i.e. the condition evaluates to true).

\textbf{type-synonym} \textit{choices} = (\textit{edge} * \textit{pState}) \ \textit{list}
— A choice is an executable edge and the process it belongs to.

\textbf{definition} \textit{getChoices} :: $gState \Rightarrow \ pState \Rightarrow \ \textit{edge} \ \textit{list} \Rightarrow \ \textit{choices}$ \textbf{where}
\begin{itemize}
\item \textit{getChoices} $g \ p = \text{foldl} (\lambda E. e)$.
if evalHandshake (cond e) (handshake g) g p ∧ evalCond (cond e) g p
then (e,p)#E
else E) []

lemma getChoices-sub-edges-fst:
  fst ' set (getChoices g p es) ⊆ set es
⟨proof⟩

lemma getChoices-sub-edges:
  (a,b) ∈ set (getChoices g p es) ⇒ a ∈ set es
⟨proof⟩

lemma getChoices-p-snd:
  snd ' set (getChoices g p es) ⊆{p}
⟨proof⟩

lemma getChoices-p:
  (a,b) ∈ set (getChoices g p es) ⇒ b = p
⟨proof⟩

definition sort-by-pri where
  sort-by-pri min-pri edges = foldl (λes e.
    let idx = nat-of-integer (abs (prio e))
    in if idx > min-pri
      then abort "Invalid priority" (λ- es)
      else let ep = e # (es ! idx) in es[idx := ep]
    ) (replicate (min-pri + 1) [] ) edges

lemma sort-by-pri-edges↓:
  assumes set edges ⊆ A
  shows set (sort-by-pri min-pri edges) ⊆ {xs. set xs ⊆ A}
⟨proof⟩

lemma sort-by-pri-edges:
  assumes set edges ⊆ A
  and es ∈ set (sort-by-pri min-pri edges)
  shows set es ⊆ A
⟨proof⟩

lemma sort-by-pri-length:
  length (sort-by-pri min-pri edges) = min-pri + 1
⟨proof⟩

definition executable
  :: states iarray ⇒ gState I ⇒ choices nres
  — Find all executable edges
  where
  executable ss g = (let procs = procs g in

64
nfoldli procs (λ- True) (λp E).
if (exclusive g = 0 ∨ exclusive g = pid p) then do {
  let (min-pri, edges) = (ss !! pState.idx p) !! pc p;
  ASSERT(set edges ⊆ edgeSet (ss !! pState.idx p));
  \( (E', -, -) \leftarrow \)
  if min-pri = 0 then do {
    WHILE \( (\lambda(E, brk, -). E = [] \land brk = 0) \) (λ (-, -, ELSE). do {
      let g = g[gState1, else := ELSE];
      E = getChoices g p edges
      in
      if E = [] then (  
        if ¬ ELSE then RETURN (E, 0::nat, True)
        else RETURN (E, 1, False)
      )
      else RETURN (E, 1, ELSE) } ([], 0::nat, False)
    )
  } else do {
    let min-pri = nat-of-integer (abs min-pri);
    let pri-edges = sort-by-pri min-pri edges;
    ASSERT (∀ es ∈ set pri-edges.
      set es ⊆ edgeSet (ss !! pState.idx p));
    let pri-edges = IArray pri-edges;
    WHILE \( (\lambda(E, pri, -). E = [] \land pri \leq \text{min-pri}) \) (λ (-, pri, ELSE).
      do {
        let es = pri-edges !! pri;
        let g = g[gState1, else := ELSE];
        let E = getChoices g p es;
        if E = [] then (  
          if ¬ ELSE then RETURN (E,pri,True)
          else RETURN (E, pri + 1, False)
        )
        else RETURN (E, pri, ELSE) } ([], 0, False)
    )
    RETURN (E'@E)
  } else RETURN E
};
}

\textbf{definition}
while-rel1 =
\text{measure } (\lambda x. \text{if } x = [] \text{ then } 1 \text{ else } 0)
<*lex*> measure (\lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } 0)
<*lex*> measure (\lambda x. \text{if } \neg x \text{ then } 1 \text{ else } 0)

\textbf{lemma} wf-while-rel1 :
wf while-rel1
(proof)

\textbf{definition}
while-rel2 mp =
\[
\text{measure } (\lambda x. \text{ if } x = \emptyset \text{ then } 1 \text{ else } 0)
\]
\[
<\text{*lex*}> \text{ measure } (\lambda x. (mp + 1) - x)
\]
\[
<\text{*lex*}> \text{ measure } (\lambda x. \text{ if } \neg x \text{ then } 1 \text{ else } 0)
\]

**lemma** \textit{wf-while-rel2}: \\
\textit{wf (while-rel2 mp)}

**lemma** \textit{executable-edgeSet}: \\
\textit{assumes} \textit{gState-inv prog g} \\\n\textit{and} \textit{program-inv prog} \\\n\textit{and} \textit{ss = states prog} \\
\textit{shows} \textit{executable ss g} \\
\leq \textit{SPEC } (\lambda cs. \forall (e,p) \in \text{ set } cs. \\
\hspace{1cm} e \in \text{ edgeSet } ((\text{states prog}) !! pState.idx p) \\
\hspace{1cm} \land pState-inv prog p \\
\hspace{1cm} \land cl-inv (g,p))

**lemma** \textit{executable-edgeSet'}: \\
\textit{assumes} \textit{gState-inv prog g} \\\n\textit{and} \textit{program-inv prog} \\
\textit{shows} \textit{executable} \textit{(states prog) g} \\
\leq \textit{SPEC } (\lambda cs. \forall (e,p) \in \text{ set } cs. \\
\hspace{1cm} e \in \text{ edgeSet } ((\text{states prog}) !! pState.idx p) \\
\hspace{1cm} \land pState-inv prog p \\
\hspace{1cm} \land cl-inv(g,p))

**schematic-lemma** \textit{executable-refine}: \\
\textit{RETURN } (\exists ex \ s \ g) \leq \textit{executable s g}

**concrete-definition** \textit{executable-impl} \textit{for s g uses} \textit{executable-refine}

5.8.3 Successor calculation

**function** \textit{to1} \textit{where} \\
\textit{to1} \textit{(} \| gState.vars = v, channels = ch, timeout = t, procs = p \|) \\
= \{ gState.vars = v, channels = ch, timeout = \text{False}, procs = p, \\
\hspace{1cm} handshake = 0, hsdata = [], exclusive = 0, gState1.else = \text{False} \}

**termination** \textit{\langle proof \rangle}

**function** \textit{from1} \textit{where} \\
\textit{from1} \textit{(} \| gState.vars = v, channels = ch, timeout = t, procs = p, \ldots = m \|) \\
= \{ gState.vars = v, channels = ch, timeout = t, procs = p \}

**termination** \textit{\langle proof \rangle}
function reset₁ where
reset₁ (gState, vars = v, channels = ch, timeout = t, procs = p,
handshake = hs, hsdata = hsd, exclusive = - , gState₁ else = - )
= (gState, vars = v, channels = ch, timeout = False, procs = p,
handshake = 0, hsdata = if hs ≠ 0 then hsd else [], exclusive = 0,
gState₁ else = False )
⟨proof⟩
termination ⟨proof⟩

lemma gState-inv-to₁:
gState-inv prog g = gState-inv prog (to₁ g) ⟨proof⟩

lemma gState-inv-from₁:
gState-inv prog g = gState-inv prog (from₁ g) ⟨proof⟩

lemma gState-inv-reset₁:
gState-inv prog g = gState-inv prog (reset₁ g) ⟨proof⟩

lemmas gState-inv-I-simps =
gState-inv-to₁ gState-inv-from₁ gState-inv-reset₁

definition removeProcs
— Remove ended processes, if there is no running one with a higher pid.
where
removeProcs ps = foldr (λp (dead, sd, ps, dcs).
  if dead ∧ pc p = 0 then (True, True, ps, pState.channels p @ dcs)
  else (False, sd, p # ps, dcs)) ps (True, False, [], [])

lemma removeProcs-subset₁:
set (fst (snd (snd (removeProcs ps)))) ⊆ set ps ⟨proof⟩

lemma removeProcs-length₁:
length (fst (snd (snd (removeProcs ps)))) ≤ length ps ⟨proof⟩

lemma removeProcs-subset:
removeProcs ps = (dead, sd, ps', dcs) ⇒ set ps' ⊆ set ps ⟨proof⟩

lemma removeProcs-length:
removeProcs ps = (dead, sd, ps', dcs) ⇒ length ps' ≤ length ps ⟨proof⟩

67
**definition** `cleanChans :: integer list ⇒ channels ⇒ channels`
— Mark channels of closed processes as invalid.

**where**

\[
\text{cleanChans } d\text{chans } cs = \text{snd (foldl } (\lambda (i, cs) c. \\
\text{ if List.member } d\text{hans } i \text{ then } (i + 1, \text{cs@[(InvChannel]})) \text{ else } (i + 1, \text{cs@c[i]})) (0, []) cs)
\]

**lemma** `cleanChans-channel-inv`:
parms `set cs ⊆ Collect channel-inv`
shows `set (cleanChans d\text{chans } cs) ⊆ Collect channel-inv`

**lemma** `cleanChans-length`:
\[\text{length (cleanChans } d\text{hans } cs) = \text{length } cs\]

**definition** `checkDeadProcs :: 'a gState-scheme ⇒ 'a gState-scheme` where

\[
\text{checkDeadProcs g } = (\\text{let } (-, \text{soDied, pros, d\text{hans} }) = \text{removeProcs } (\text{pros } g) \text{ in } \\
\text{if soDied then } \\
\text{g\{|pros := pros, channels := cleanChans } d\text{hans } (\text{channels } g)\|} \\
\text{else } g)
\]

**lemma** `checkDeadProcs-gState-progress-rel`:
parms `g\text{State-inv prog } g`
shows `(g, checkDeadProcs } g) \in g\text{State-progress-rel prog}

**lemma** `g\text{State-progress-rel-exclusive}`:
\[\langle g, g\rangle \in g\text{State-progress-rel prog} \implies (g, g\{|exclusive := p\}) \in g\text{State-progress-rel prog}\]

**definition** `apply\text{Edge} :: program ⇒ edge ⇒ p\text{State} ⇒ g\text{State}_1 ⇒ g\text{State}_1 n\text{res}`
wheres` apply\text{Edge} prog e p g = do \\

\[
\text{let } (g', p') = \text{eval\text{Effect} } (\text{effect } e) \text{ prog } g p; \\
\text{ASSERT } ((g, g') \in g\text{State-progress-rel prog}); \\
\text{ASSERT } (p\text{State-inv prog } p'); \\
\text{ASSERT } (cl\text{-inv } (g', p'));
\]

\[
\text{let } p'' = \text{case target } e \text{ of Index } t \Rightarrow \\
\text{if } t < \text{ IArray.length } (\text{states } prog \text{!! } p\text{State}.idx p') \text{ then } p'\{|p\text{c} := t\} \\
\text{else abort "Edge target out of bounds" } (\lambda -. p') \\
\text{- ⇒ abort "Edge target not Index" } (\lambda -. p'); \\
\text{ASSERT } (p\text{State-inv prog } p'');
\]

68
let \(g'\) = list-update \((\text{procs} := g')\) \((\text{pid} p'' - 1)\) p'';

\(\text{ASSERT} \ ((g', g'') \in \text{gState-progress-rel prog});\)

let \(g'''\) = \((\text{if isAtomic } e \land \text{handshake } g'' = 0\) \\
\(\text{then } g''\) exclusive := pid p'' \)
\(\text{else } g''\);

\(\text{ASSERT} \ ((g', g''') \in \text{gState-progress-rel prog});\)

let \(g_f = (\text{if pc } p'' = 0 \text{ then checkDeadProcs } g''\) else g''');

\(\text{ASSERT} \ ((g''', g_f) \in \text{gState-progress-rel prog});\)

\(\text{RETURN } g_f \}

\textbf{lemma} applyEdge-gState-progress-rel:
\textbf{assumes} program-inv prog \n\textbf{and} gState-inv prog g \n\textbf{and} pState-inv prog p \n\textbf{and} cl-inv \((g, p)\) 
\textbf{and} \(e \in \text{edgeSet } (\text{states prog}!!pState.idx p)\)
\textbf{shows} applyEdge prog e p g \leq \text{SPEC } (\lambda g'. (g, g') \in \text{gState-progress-rel prog})
\textit{⟨proof⟩}

\textbf{schematic-lemma} applyEdge-refine:
\textbf{RETURN} (?)ac prog e p g \leq \text{applyEdge prog e p g}
\textit{⟨proof⟩}

\textbf{concrete-definition} applyEdge-impl for e p g uses applyEdge-refine

\textbf{definition} nexts
\(:: \text{program} \Rightarrow \text{gState} \Rightarrow \text{gState ls nres}\)
— The successor function

\textbf{where}
nexts prog g = ( 
\textbf{let} 
\(f = \text{from}_1;\) 
\(g = \text{to}_1 g\) 
\textbf{in}

\(\text{REC } (\lambda D g. \text{do}\{\)} 
\(E \leftarrow \text{executable } (\text{states prog} g;\) 
\(\text{if } E = [] \text{ then}\) 
\(\text{if handshake } g \neq 0 \text{ then}\) 
\(\text{(* HS not possible -- remove current step *)}\) 
\(\text{RETURN } (\text{ls.empty}())\) 
\(\text{else if exclusive } g \neq 0 \text{ then}\) 
\(\text{(* Atomic blocks -- just return current state *)}\) 
\(\text{RETURN } (\text{ls.sng } (f g))\) 
\(\text{else if } \neg \text{timeout } g \text{ then}\) 
\(\text{(* Set timeout *)}\) 
\(D \text{ (g[]timeout := True)}}))\)
else
  (* If all else fails: stutter *)
  RETURN (ls.sng (f (reset_1 g)))

else
  (*
  Setting the internal variables (exclusive, handshake, ...) to 0
  is safe — they are either set by the edges, or not thought
  to be used outside executable.. *
  *)
  let g = reset_1 g in
  nfoldli E (λ-. True) (λ(e,p) G.
    applyEdge prog e p >>= (λ g'.
      if handshake g' ≠ 0 ∨ isAtomic e then do {
        G_R ← D g';
        if ls.isEmpty G_R ∧ handshake g' = 0 then
          (* this only happens if the next step is a handshake, which fails
          hence we stay at the current state *)
          RETURN (ls.ins (f g') G)
        else
          RETURN (ls.union G_R G)
        } else RETURN (ls.ins (f g') G)) (ls.empty())
  ) g
  >>= (λG. if ls.isEmpty G then RETURN (ls.sng (f g)) else RETURN G)

lemma gState-progress-rel-intros:
  (to_1 g, gI') ∈ gState-progress-rel prog
  ⇒ (g, from_1 gI') ∈ gState-progress-rel prog
  (gI, gI') ∈ gState-progress-rel prog
  ⇒ (gI, reset_1 gI') ∈ gState-progress-rel prog
  (to_1 g, gI') ∈ gState-progress-rel prog
  ⇒ (to_1 g, gI'(timeout := t)) ∈ gState-progress-rel prog
⟨proof⟩

lemma gState-progress-rel-step-intros:
  (to_1 g, g') ∈ gState-progress-rel prog
  ⇒ (reset_1 g', g'') ∈ gState-progress-rel prog
  ⇒ (g, from_1 g'') ∈ gState-progress-rel prog
  (to_1 g, g') ∈ gState-progress-rel prog
  ⇒ (reset_1 g', g'') ∈ gState-progress-rel prog
  ⇒ (to_1 g, g'') ∈ gState-progress-rel prog
⟨proof⟩

lemma cl-inv-reset_1:
  cl-inv(g,p) ⇒ cl-inv(reset_1 g, p)
⟨proof⟩

lemmas refine-helpers =
gState-progress-rel-intros gState-progress-rel-step-intros cl-inv-reset_1
lemma nexts-SPEC:
  assumes gState-inv prog g
  and program-inv prog
  shows nexts prog g \leq SPEC (\lambda gs. \forall g' \in ls. \alpha gs. (g,g') \in gState-progress-rel prog)
  (proof)

lemma RETURN-dRETURN:
  RETURN f \leq f' \implies nres-of (dRETURN f) \leq f'
  (proof)

lemma executable-dRETURN:
  nres-of (dRETURN (executable-impl prog g)) \leq executable prog g
  (proof)

lemma applyEdge-dRETURN:
  nres-of (dRETURN (applyEdge-impl prog e p g)) \leq applyEdge prog e p g
  (proof)

schematic-lemma nexts-code-aux:
  nres-of (?nexts prog g) \leq nexts prog g
  (proof)

concrete-definition nexts-code-aux for prog g uses nexts-code-aux
prepare-code-thms nexts-code-aux-def

5.8.4 Handle non-termination

A Promela model may include non-terminating parts. Therefore we cannot guarantee, that nexts will actually terminate. To avoid having to deal with this in the model checker, we fail in case of non-termination.

definition SUCCEED-abort where
SUCCEED-abort msg dm m = ( 
  case m of
  RES X \Rightarrow if X={} then Code.abort msg (\lambda dm) else RES X
  | - \Rightarrow m)

definition dSUCCEED-abort where
dSUCCEED-abort msg dm m = ( 
  case m of
  dSUCCEDi \Rightarrow Code.abort msg (\lambda dm)
  | - \Rightarrow m)

definition ref-succeed where
ref-succeed m m' \leftrightarrow m \leq m' \land (m=SUCCEED \implies m'=SUCCEED)
\textbf{lemma} dSUCCEED-abort-SUCCEED-abort: 
\[ \begin{array}{l} \text{RETURN } dm' \leq dm; \text{ref-succeed} \ (\text{ners-of } m') \ m \end{array} \]
\[ \implies ners-of \ (dSUCCEED-abort \ msg \ (d\text{return} \ dm) \ (m')) \leq SUCCEED-abort \ msg \ dm \ m \]

(proof)

The final successor function now incorporates:

1. \textit{nexts}

2. handling of non-termination

\textbf{definition} \textit{nexts-code} \textbf{where}
\[ \text{nexts-code prog } g = \]
\[ \text{the-res} \ (dSUCCEED-abort \ ((\text{str} "The Universe is broken!")) \ (d\text{return} \ (ls.sng \ g)) \ (\text{nexts-code-aux prog } g)) \]

\textbf{lemma} \textit{nexts-code-SPEC}:
\[ \text{assumes } g\text{State-inv prog } g \]
\[ \text{and program-inv prog} \]
\[ \text{shows } g' \in \text{ls.\alpha} \ (\text{nexts-code prog } g) \]
\[ \implies (g,g') \in g\text{State-progress-rel prog} \]

(proof)

5.9 Finiteness of the state space

\textbf{inductive-set} \textit{reachable-states} \textbf{for} \[ P :: \text{program} \]
\[ \text{and } g_s :: \text{gState} — \text{start state} \]
\[ \text{where} \]
\[ g_s \in \text{reachable-states } P \ g_s | \]
\[ g \in \text{reachable-states } P \ g_s \implies x \in \text{ls.\alpha} \ (\text{nexts-code } P \ g) \]
\[ \implies x \in \text{reachable-states } P \ g_s \]

\textbf{lemmas} \textit{reachable-states-induct\{case-names init step\} =}
\[ \text{reachable-states.induct\{split-format (complete)\}} \]

\textbf{lemma} \textit{reachable-states-finite}:
\[ \text{assumes program-inv prog} \]
\[ \text{and gState-inv prog } g \]
\[ \text{shows finite (reachable-states prog } g) \]

(proof)

5.10 Traces

When trying to generate a lasso, we have a problem: We only have a list of global states. But what are the transitions to come from one to the other? This problem shall be tackled by \textit{replay}: Given two states, it generates a list of transitions that was taken.
**Definition** replay :: program \(\Rightarrow\) gState \(\Rightarrow\) gState \(\Rightarrow\) choices nres

where

replay prog g₁ g₂ = (let
    g₁ = to₁ g₁;
    check = λg. from₁ g = g₂
in
    REC \(\lambda\)₆ \(g\). do {E ← executable (states prog) g;
      if E = [] then
         if check g then RETURN []
         else if ¬ timeout g then D (g∥timeout := True))
         else abort "Stuttering should not occur on replay"
           (λ-. RETURN [])
else
    let g = reset₁ g in
    nfoldli E (λE. E = []) (λ(e,p) -.
      applyEdge prog e p g \(\Rightarrow\) (λg'.
        if handshake g' \(\neq\) 0 \(\lor\) isAtomic e then do {
          Eᵣ ← D g';
          if Eᵣ = [] then
             if check g' then RETURN [(e,p)] else RETURN []
           else
             RETURN ((e,p) \# Eᵣ)
        }))
    ) []
})) g₁)

**Lemma** abort-refine[refine-transfer]:

\[ \text{nres-of } (f ()) \leq F () \implies \text{nres-of } (\text{abort } s f) \leq \text{abort } s F \]

\[ f() \neq d\text{SUCCEED} \implies \text{abort } s f \neq d\text{SUCCEED} \]

-proof\]

**Schematic-Lemma** replay-code-aux:

\[ \text{RETURN } (?\text{replay prog } g₁ g₂) \leq \text{replay prog } g₁ g₂ \]

-proof\]

**Concrete-Definition** replay-code for prog g₁ g₂ uses replay-code-aux

**Prepare-Code-Thms** replay-code-def

### 5.10.1 Printing of traces

**Definition** procDescr

\[ :: (\text{integer }\Rightarrow\text{string}) \Rightarrow\text{program }\Rightarrow\text{pState }\Rightarrow\text{string} \]

where

\[ \text{procDescr } f \text{ prog } p = (\text{let}
    \text{name} = \text{String.expplode (proc-names prog }!! \text{pState}.\text{idx }p);\]
    \text{id} = f (\text{integer-of-nat (pid }p))\]
definition printInitial :: (integer ⇒ string) ⇒ program ⇒ gState ⇒ string
where
  printInitial f prog g 0 = (let psS = printList (procDescr f prog) (procs g 0) [] [] "" "" in ""Initially running: "$ & psS)

abbreviation lf ≡ Char Nibble0 NibbleA

fun printConfig :: (integer ⇒ string) ⇒ program ⇒ gState option ⇒ gState ⇒ string
where
  printConfig f prog None g 0 = printInitial f prog g
  | printConfig f prog (Some g0) g1 = (let eps = replay-code prog g0 g1 in let print = (λ(e,p). procDescr f prog p @ "." @ printEdge f (pc p) e) in if eps = [] ∧ g1 = g0 then "" −− stutter −−"" else printList print eps [] (lf # "" "")

definition printConfigFromAST f ≡ printConfig f o fst o setUp

5.11 Code export

code-identifier
code-module PromelaInvariants ⇒ (SML) Promela
| code-module PromelaDatastructures ⇒ (SML) Promela

definition executable-triv prog g = executable-impl (snd prog) g
definition apply-triv prog g ep = applyEdge-impl prog (fst ep) (snd ep) (reset_I g)

export-code
setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv extractLTLs lookupLTL
checking SML

export-code
setUp printProcesses printConfigFromAST nexts-code executable-triv apply-triv extractLTLs lookupLTL
in SML
| file Promela.sml

end

6 LTL integration

theory PromelaLTL
We have a semantic engine for Promela. But we need to have an integration with LTL – more specifically, we must know when a proposition is true in a global state. This is achieved in this theory.

### 6.1 LTL optimization

For efficiency reasons, we do not store the whole `expr` on the labels of a system automaton, but `nat` instead. This index then is used to look up the corresponding `expr`.

```plaintext
type-synonym APs = expr iarray

primrec ltlc-aps-list' :: 'a ltlc ⇒ 'a list ⇒ 'a list
where
  ltlc-aps-list' LTLcTrue  l = l
  | ltlc-aps-list' LTLcFalse l = l
  | ltlc-aps-list' (LTLcProp p) l = (if List.member l p then l else p#l)
  | ltlc-aps-list' (LTLcNeg x) l = ltlc-aps-list' x l
  | ltlc-aps-list' (LTLcNext x) l = ltlc-aps-list' x l
  | ltlc-aps-list' (LTLcFinal x) l = ltlc-aps-list' x l
  | ltlc-aps-list' (LTLcGlobal x) l = ltlc-aps-list' x l
  | ltlc-aps-list' (LTLcAnd x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
  | ltlc-aps-list' (LTLcOr x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
  | ltlc-aps-list' (LTLcImplies x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
  | ltlc-aps-list' (LTLcIff x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
  | ltlc-aps-list' (LTLcUntil x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)
  | ltlc-aps-list' (LTLcRelease x y) l = ltlc-aps-list' y (ltlc-aps-list' x l)

lemma ltlc-aps-list' -correct:
  set (ltlc-aps-list' φ l) = ltlc-aprops φ ∪ set l
  (proof)

lemma ltlc-aps-list' -distinct:
  distinct l ⇒ distinct (ltlc-aps-list' φ l)
  (proof)

definition ltlc-aps-list :: 'a ltlc ⇒ 'a list
where
  ltlc-aps-list φ = ltlc-aps-list' φ []

lemma ltlc-aps-list -correct:
  set (ltlc-aps-list φ) = ltlc-aprops φ
  (proof)
```
lemma ltlc-aps-list-distinct:
  distinct (ltlc-aps-list ϕ)
⟨proof⟩

primrec idx' :: nat ⇒ 'a list ⇒ 'a ⇒ nat option where
  idx' [] = None
| idx' ctr (x#xs) y = (if x = y then Some ctr else idx' (ctr+1) xs y)
definition idx = idx' 0

lemma idx'-correct:
  assumes distinct xs
  shows idx' ctr xs y = Some n ↔ n ≥ ctr ∧ n < length xs + ctr ∧ xs ! (n-ctr) = y
⟨proof⟩

lemma idx-correct:
  assumes distinct xs
  shows idx xs y = Some n ↔ n < length xs ∧ xs ! n = y
⟨proof⟩

lemma idx-dom:
  assumes distinct xs
  shows dom (idx xs) = set xs
⟨proof⟩

lemma idx-image-self:
  assumes distinct xs
  shows (the ◦ idx xs) ' set xs = {..<length xs}
⟨proof⟩

lemma idx-ran:
  assumes distinct xs
  shows ran (idx xs) = {..<length xs}
⟨proof⟩

lemma idx-inj-on-dom:
  assumes distinct xs
  shows inj-on (idx xs) (dom (idx xs))
⟨proof⟩

definition ltl-convert :: expr ltlc ⇒ APs × nat ltlc where
  ltl-convert ϕ = (let APs = ltlc-aps-list ϕ;
                   ϕ_1 = map-ltlc (the ◦ idx APs) ϕ
                   in (IArray APs, ϕ_1))

lemma ltl-convert-correct:
  assumes ltl-convert ϕ = (APs, ϕ_1)
shows \texttt{ltlc-aprops} $\phi = \text{set (IArray.list-of APs)}$ (is \(\phi P1\))
and \texttt{ltlc-aprops} $\phi_i = \{..<\text{IArray.length APs}\}$ (is \(\phi P2\))
and $\phi_i = \text{map-ltlc (the o idx (IArray.list-of APs))} \phi$ (is \(\phi P3\))
and distinct (IArray.list-of APs)
(\emph{proof})

definition prepare
\[:: - \times (\text{program} \Rightarrow \text{unit}) \Rightarrow \text{ast} \Rightarrow \text{expr ltlc} \Rightarrow (\text{program} \times \text{APs} \times \text{gState}) \times \text{nat ltlc}\]
where
prepare cfg ast $\phi$ \equiv
\hspace{1em} let
\hspace{2em} (prog, $g_0$) = Promela.setUp ast;
\hspace{2em} (APs, $\phi_i$) = PromelaLTL.ltl-convert $\phi$
\hspace{2em} in
\hspace{3em} ((prog, APs, $g_0$), $\phi_i$)

lemma prepare-instrument[code]:
prepare cfg ast $\phi$ \equiv
\hspace{1em} let
\hspace{2em} (-, printF) = cfg;
\hspace{2em} - = PromelaStatistics.start ();
\hspace{2em} (prog, $g_0$) = Promela.setUp ast;
\hspace{2em} - = printF prog;
\hspace{2em} (APs, $\phi_i$) = PromelaLTL.ltl-convert $\phi$;
\hspace{2em} - = PromelaStatistics.stop-timer ()
\hspace{2em} in
\hspace{3em} ((prog, APs, $g_0$), $\phi_i$)
(\emph{proof})

export-code prepare checking SML

6.2 Language of a Promela program

definition propValid :: $\text{APs} \Rightarrow g\text{State} \Rightarrow \text{nat} \Rightarrow \text{bool}$ where
propValid APs $g$ i \begin{align*}
&= i < \text{IArray.length APs} \wedge \\
&\text{exprArith g emptyProc (APs!!i)} \\
&\neq 0
\end{align*}
definition promela-E :: $\text{program} \Rightarrow (\text{gState} \times \text{gState})$ set
— Transition relation of a promela program
where
promela-E prog \equiv\{($(g,g')$, $g' \in \text{ls.\alpha}$ (nexts-code prog $g$))\}
definition promela-E-ll \:: \text{program} \times \text{APs} \Rightarrow (\text{gState} \times \text{gState})$ set where
promela-E-ll = promela-E \circ \text{fst}
definition promela-is-run' :: $\text{program} \times g\text{State} \Rightarrow g\text{State word} \Rightarrow \text{bool}$
— Predicate defining runs of promela programs
where
promela-is-run′ progg r ≡
  let (prog,gs₀) = progg in
  r ₀ = gs₀
  ∧ (∀ i. r (Suc i) ∈ ls.ₐ (nexts-code prog (r i)))

abbreviation promela-is-run ≡ promela-is-run′ o setUp

definition promela-is-run-ltl :: program × APs × gState ⇒ gState word ⇒ bool
  where
    promela-is-run-ltl promg r ≡ let (prog,APs,g) = promg in promela-is-run′ (prog,g)

definition promela-props :: gState ⇒ expr set
  where
    promela-props g = { e. exprArith g emptyProc e ≠ 0 }

definition promela-props-ltl :: APs ⇒ gState ⇒ nat set
  where
    promela-props-ltl APs g ≡ Collect (propValid APs g)

definition promela-language :: ast ⇒ expr set word set
  where
    promela-language ast ≡ { promela-props o r | r. promela-is-run ast r }

definition promela-language-ltl :: program × APs × gState ⇒ nat set word set
  where
    promela-language-ltl promg ≡ let (prog,APs,g) = promg in
        { promela-props-ltl APs o r | r. promela-is-run-ltl promg r }

lemma promela-props-ltl-map-aprops:
  assumes ltl-convert ϕ = (APs,ϕ₁)
  shows promela-props-ltl APs =
        map-aprops (idx (IArray.list-of APs)) o promela-props

⟨ proof ⟩

lemma promela-run-in-language-iff:
  assumes conv: ltl-convert ϕ = (APs,ϕ₁)
  shows promela-props o ϕ ∈ ltlc-language ϕ
        ⇔ promela-props-ltl APs o ϕ ∈ ltlc-language ϕ₁ (is ?L ⇔ ?R)

⟨ proof ⟩

lemma promela-language-sub-iff:
  assumes conv: ltl-convert ϕ = (APs,ϕ₁)
  and setUp: setUp ast = (prog,g)
  shows promela-language-ltl (prog,APs,g) ⊆ ltlc-language ϕ₁ ⇔ promela-language ast ⊆ ltlc-language ϕ

⟨ proof ⟩
hide-const (open) abort abort' abortv
     err err' errv
     usc usc'
     warn the-warn with-warn

hide-const (open) idx idx'
end
theory PromelaLTLConv
imports
  Promela
  ../LTL-to-GBA/LTL
begin

6.3 Proposition types and conversion

LTL formulae and propositions are also generated by an SML parser. Hence we have the same setup as for Promela itself: Mirror the data structures and (sometimes) map them to new ones.

This theory is intended purely to be used by frontend code to convert from propc to expr. The other theories work on expr directly. While we could of course convert directly, that would introduce yet a semantic level.

datatype binOp = Eq | Le | LEq | Gr | GEq

datatype ident = Ident String.literal integer option

datatype propc = CProp ident
  | BProp binOp ident ident
  | BExpProp binOp ident integer

fun identConv :: ident ⇒ varRef where
  identConv (Ident name None) = VarRef True name None
  | identConv (Ident name (Some i)) = VarRef True name (Some (ExprConst i))

definition ident2expr :: ident ⇒ expr where
  ident2expr = ExprVarRef ◦ identConv

primrec binOpConv :: binOp ⇒ PromelaDatastructures.binOp where
  binOpConv Eq = BinOpEq
  | binOpConv Le = BinOpLe
  | binOpConv LEq = BinOpLEq
  | binOpConv Gr = BinOpGr
  | binOpConv GEq = BinOpGEq

primrec propc2expr :: propc ⇒ expr where
  propc2expr (CProp ident) =
      ExprBinOp BinOpEq (ident2expr ident) (ExprConst 1)
| propc2expr (BProp bop il ir) = 
  ExprBinOp (binOpConv bop) (ident2expr il) (ident2expr ir) |
| propc2expr (BExpProp bop il ir) = 
  ExprBinOp (binOpConv bop) (ident2expr il) (ExprConst ir) |

definition ltl-conv :: propc ltlc ⇒ expr ltlc where
  ltl-conv = map-ltlc propc2expr

definition printPropc :: (integer ⇒ char list) ⇒ propc ⇒ char list
  where
    printPropc f p = printExpr f (propc2expr p)

The semantics of a propc is given just for reference.

definition evalPropc :: gState ⇒ propc ⇒ bool where
  evalPropc g p ←→ exprArith g emptyProc (propc2expr p) ≠ 0

end

References
