LoopW
Technical Reference
version 0.3

BY EMMANUEL POLONOWSKI

December 20, 2009

Abstract

This document describes the implementation of the LoopW language, an imperative language with higher-order procedural variables and non-local jumps equiped with a program logic, in SML. It includes the user manual along with some implementation notes and many examples of certified imperative programs. As a concluding example, we show the certification of an imperative program encoding shift/reset using callcc/throw and a global meta-continuation.

1 Introduction

This document describes the implementation in Standard ML of the LoopW language [2, 1], an imperative language with higher-order procedural variables and non-local jumps equiped with a program logic.

This implementation consists firstly in a type inference tool prototype `LoopW` that reads partially annotated imperative source code and performs combinated type checking and type inference; and secondly in a proof checking tool prototype `LoopWc` that reads completely annotated imperative source code and performs proof checking, translation into completely annotated functional source code and proof checking of the functional code.

The complete source code archive of those prototypes along with numerous small examples of certified programs can be obtained at `http://lacl.univ-paris12.fr/polonowski/`.

Section 4 contains the user manual, describing the source syntax, the usage of the tools `LoopW` and `LoopWc`, and a small introduction on imperative program certification with examples. Section 4 gives details about the current implementation, along with small commented protions of Standard ML source code. Section 4 presents the overview of the source code documentation (generated using `smldoc` – from SML# – `http://www.pllab.riec.tohoku.ac.jp/smlsharp/`) which is included in the source code archive.

Bibliography

[1] Tristan Crolard and Emmanuel Polonowski. A program logic for higher-order procedural variables and non-local jumps. 2010. Submitted.

[2] Tristan Crolard, Emmanuel Polonowski, and Pierre Valarcher. Extending the loop language with higher-order procedural variables. *ACM Transactions on Computational Logic*, 10(4):1–37, 08 2009.

2 User Manual

2.1 Source syntax

```
(terms)        t ::= i                                     term variable
                   | f(t, ..., t)                          function application
                   | 0                                     zero
                   | s(t)                                  successor
                   | p(t)                                  predecessor
                   | t + t                                 addition
                   | t * t                                 multiplication
                   | t - t                                 substraction
```

```
(types)        T ::= $                                     absurd
                   | (t = t)                               equality over terms
                   | nat(t)                                nat predicate
                   | proc({i, ..., i} in T, ..., T;        procedure prototype
                          {i, ..., i} out T, ..., T)
                   | ~T                                    negation
```

```
(expressions)  e ::= q̄                                     natural numbers constants
                   | X                                     program variable
                   | *                                     true value
                   | proc({i, ..., i} in X:T, ..., X:T;    anonymous procedure declaration
                          {i, ..., i} out X:T, ..., X:T) { s }
```

```
(commands)     c ::= { s }X:T, ..., X:T                    annotated block
                   | for I := 0 until e { s }X:T, ..., X:T annotated for loop
                   | X := e                                assignment
                   | inc(X)                                incrementation
                   | dec(X)                                decrementation
                   | e(e, ..., e; X, ..., X)               procedure call
                   | X : { s }X:T, ..., X:T                labeled block
                   | jump(e, e, ..., e)X:T, ..., X:T       jump to a label
```

```
(sequences)    s ::= ε                                     empty sequence
                   | c; s                                  command composition
                   | cst X = e; s                          constant declaration
                   | var X := e; s                         variable declaration
```

2.2 Usage

```
========== LoopW imperative proof inference and proof checking program ==========

This tool contains two executables, namely loopW and loopWc.

I. loopW: imperative proof inference.

The loopW inference tool reads its input in a file with extension ".loop" and
tries to infer a correct and complete proof derivation that corresponds to the
given source code.

usage: loopW [-version] [-v123] [-uprint] [-print] [-pprint] [-form] [-tex] [-
ott] file.loop
  -version print version and exit
  -v Verbosity level (1 : low, 2 : medium, 3 : high)
  -uprint Erase all type information and pretty print (file.cs)
  -print Pretty print (file.rev.loop)
  -pprint Complete infered proof pretty printing (file.proof)
  -form Display formula-like types instead of imperative types
  -tex Tex output
  -ott Ott/twelf complete infered proof pretty printing (file.loop.ott)

NOTES
- On Unix systems (including Mac OS X) you need to have mono (www.mono-
project.com)
  installed and you can then run the command as:

  mono loopW.exe [-version] [-v123] [-uprint] [-print] [-pprint] [-form] [-tex]
[-ott] file.loop

II. loopWc: imperative proof checking, functional translation and proof checking.

The loopWc proof checker and translater tool reads its input in a file with
extension ".proof" (eventually written by the loopW tool (see above)). It first
checks the imperative proof derivation, then it translate it into a functional
proof derivation and check it.

usage: loopWc [-version] [-v123] [-uprint] [-fprint] file.proof
```

```
  -version print version and exit
  -v Verbosity level (1 : low, 2 : medium, 3 : high)
  -uprint Erase all type information and pretty print (file.pcs)
  -fprint Functional translation (untyped) pretty print (file.sml)

NOTES
- On Unix systems (including Mac OS X) you need to have mono (www.mono-
project.com)
  installed and you can then run the command as:

  mono loopWc.exe [-version] [-v123] [-uprint] [-fprint] file.proof

- The generated sml file can be tested with SML/NJ with the provided library
file:

  sml lib.sml file.sml
```

2.3 Program certification examples

2.3.1 Addition

```
cst p_add = proc({x, y} in X:nat(x), Y:nat(y); out Z:nat(x + y)) {
    Z := X :> nat(x + 0);
    for i := 0 until Y {
        inc(Z);
    }Z:nat(x + i);
};
var N := *;
p_add(3, 5; N);
```

To compile this example (in a file `add.loop`), we use the command: `loopW add.loop`. The absence of output indicates that the compilation was successfull. We find a new file `add.typ` containing the following text:

```
cst p_add = proc(in X, Y; out Z) {
    Z := X;
    for i := 0 until Y {
        inc(Z);
    }Z;
};
var N := *;
p_add(3, 5; N);
```

```
-(X:nat(x), Y:nat(y))[Z:(0 = 0)]
|   [Z:nat((x + 0))]    by #1
|   -(i:nat(i))[Z:nat((x + i))]
|   |    [Z:nat(s((x + i)))]
|   [Z:nat((x + y))]    by #2
(p_add:proc({x, y} in nat(x), nat(y); out nat((x + y))))
[N:(0 = 0)]
[N:nat((3 + 5))]
```

```
1: |- (x = (x + 0))
2: |- (s((x + i)) = (x + s(i)))
```

We find on the left part our source code (without type informations) and on the right part the typing derivation. Remark that variables may change their type: `N` is declared and assigned with `*` of type `0 = 0`, and after the procedure call, its type is `nat(3 + 5)`.

The bottom part of the file contains the equalities used to certify the source code. They must be checked (either manually or with a solver) in order to guarantee the correctness of the code. Here, the equalities are straightforward consequences of the usual axioms defining addition for peano arithmetic.

In order to proof-check this program, we need to generate the intermediate file `add.proof`. We use the command `loopW -pprint add.loop`. We can then call the proof-checker: `loopWc add.proof`. Again, the absence of output indicates that the proof-checking was successfull. We can also generate the functional program: `loopWc -fprint add.proof`. It gives us the following file `add.sml`:

```
let val p_add (* : Forall x,y.((nat(x) & nat(y)) => (nat((x + y)))) *) = fn (X (* : nat(x) *), Y (*
: nat(y) *)) =>
      let val Z (* : nat((x + 0)) *) = X (* : nat(x) *) (* :> {var_1/nat(var_1)}[x = (x + 0) by #1]
*)
          val Z (* : nat((x + y)) *) = Rec (Y (* : nat(y) *), Z (* : nat((x + 0)) *), fn i (* :
nat(i) *) => fn (Z (* : nat((x + i)) *)) =>
            let val Z (* : nat(s((x + i))) *) = Succ(Z (* : nat((x + i)) *))
            in Z (* : nat(s((x + i))) *) (* :> {var_2/(nat(var_2))}[s((x + i)) = (x + s(i)) by #2]
*) end )
      in Z (* : nat((x + y)) *) end
    val N (* : 0 = 0 *) = ()
    val N (* : nat((3 + 5)) *) = p_add (* : Forall x,y.((nat(x) & nat(y)) => (nat((x + y)))) *)((3,
5)) (* [3, 5] *)
in () end

(* 1: |- x = (x + 0) *)
(* 2: |- s((x + i)) = (x + s(i)) *)
```

If we erase all the comments containing the typing informations, we get:

```
let val p_add = fn (X, Y) =>
      let val Z = X
          val Z = Rec (Y , Z, fn i => fn (Z) =>
            let val Z = Succ(Z)
            in Z end)
      in Z end
    val N = ()
    val N = p_add (3, 5)
in () end
```

It clearly corresponds to a valid definition of addition using a recursion operator.

2.3.2 Ackermann

```
cst ack = proc ({m,n} in M : nat(m), N : nat(n); out Z : nat(a(m,n))) {
    var G := proc ({y} in Y : nat(y); out P : nat(a(0,y))) {
        P := Y;
        inc(P);
    };
    for i := 0 until M {
        cst H = G;
        G := proc ({y} in Y : nat(y); out P : nat(a(s(i),y))) {
            P := 2 :>  nat(a(s(i),0));
            for j := 0 until Y {
                H(P; P);
            }P:nat(a(s(i),j));
        };
    }G:proc ({y} in nat(y); {} out nat(a(i,y)));
    G(N; Z);
};
```

2.3.3 Negation

Using the negation, we can use the full power of classical logic to prove theorems. For instance, the following program is a proof of $\neg(\forall x.\mathrm{nat}(x) \wedge \neg F) \Rightarrow \exists x.\mathrm{nat}(x) \wedge F$ which is not provable in intuitionnistic logic.

```
cst dblNegElim = proc (in K : ~~F; out Z : F) {
    K2: {
        jump(K,K2)Z:F;
    }Z:F;
};
cst notAllNot_implies_Exist = proc (in H : ~proc({x} in nat(x); out ~F);
                                    out C : proc(; {x} out nat(x), F)) {
    K: {
        cst P = proc ({x} in N : nat(x); out Z : ~F) {
            K2: {
                dblNegElim(K2; Z);
                cst V = Z;
                cst Q = proc(; {x} out M : nat(x), Y : F) {
                    M := N;
                    Y := V;
                };
                jump(K, Q)Z:~F;
            }Z:~F;
        };
        jump(H, P)C:proc(; {x} out nat(x), F);
    }C:proc(; {x} out nat(x), F);
};
```

2.3.4 Shift/reset

```
cst shift = proc (in p : proc(in proc({n} in nat(n), ~A; out nat(F32(n)), ~A),
                              ~proc ({n} in nat(n), ~A; out nat(F32(n)), ~A);
                           out proc ({n} in nat(n), ~A; out nat(F32(n)), ~A),
                               ~proc ({n} in nat(n), ~A; out nat(F32(n)), ~A)),
                     mk2 : ~proc ({n} in nat(n), ~A; out nat(F32(n)), ~A);
                  {u} out r : nat(u), mk : ~nat(F32(u))) {
    mk := mk2;
    cst reset2 = proc ({x} in p : proc(in ~nat(F32(x)); out H,~H), mk2 : ~A;
                       out r : nat(F32(x)), mk : ~A) {
        mk := mk2;
        k: {
            cst m = mk;
            mk := proc (in r : nat(F32(x)); out Z : $) {
                jump(k, r, m)Z:$;
            };
            var y := *;
            p(mk; y, mk);
            jump(mk, y)r:nat(F32(x)), mk:~A;
        }r:nat(F32(x)), mk:~A;
    };
    k: {
        cst q = proc ({x} in v : nat(x), mk2 : ~A; out r : nat(F32(x)), mk : ~A) {
            mk := mk2;
            cst anonym = proc(in mk2 : ~nat(F32(x)); out z : H, mk : ~H) {
                mk := mk2;
                jump(k, v, mk)z:H,mk:~H;
            };
            reset2(anonym, mk; r, mk);
        };
        var y := *;
        p(q, mk; y, mk);
```

```
        jump(mk, y)r:nat(0), mk:~nat(F32(0));
    }{u}r:nat(u), mk:~nat(F32(u));
};
cst reset = proc (in p : proc(in ~proc({n} in nat(n), ~A; out nat(F32(n)), ~A);
                              {v} out nat(v), ~nat(v)), mk2 : ~A;
                  out r : proc({n} in nat(n), ~A; out nat(F32(n)), ~A), mk : ~A) {
    mk := mk2;
    k: {
        cst m = mk;
        mk := proc (in r : proc({n} in nat(n), ~A; out nat(F32(n)), ~A); out Z : $) {
            jump(k, r, m)Z:$;
        };
        var y := *;
        p(mk; y, mk);
        jump(mk, y)r:proc({n} in nat(n), ~A; out nat(F32(n)), ~A), mk:~A;
    }r:proc({n} in nat(n), ~A; out nat(F32(n)), ~A), mk:~A;
};
cst a = proc (in mk2 : ~A; out z : nat(3+2), mk : ~A) {
    cst p_add = proc ({x,y} in X : nat(x), Y : nat(y), mk2 : ~A; out Z : nat(x + y), mk : ~A) {
        mk := mk2;
        Z := X :> nat(x + 0);
        for i := 0 until Y {
            inc(Z);
        }Z:nat(x + i);
    };
    cst q = proc(in mk2 : ~proc({n} in nat(n), ~A; out nat(F32(n)), ~A);
                 {v} out r : nat(v), mk : ~nat(v)) {
        mk := mk2;
        cst p = proc(in f : proc ({n} in nat(n), ~A; out nat(F32(n)), ~A),
                        mk2 : ~proc ({n} in nat(n), ~A; out nat(F32(n)), ~A);
                     out h : proc ({n} in nat(n), ~A; out nat(F32(n)), ~A),
                         mk : ~proc ({n} in nat(n), ~A; out nat(F32(n)), ~A)) {
            mk := mk2;
            h := f;
        };
        var b := *;
        shift(p, mk; b, mk);
        r := 3 :> nat(F32(0));
        for i := 0 until b {
            r := 2 :> nat(F32(s(i)));
        }r:nat(F32(i));
    };
    var mk := mk2;
    var g := *;
    reset(q, mk; g, mk);
    var x := *;
    g(0, mk; x, mk);
    var y := *;
    g(1, mk; y, mk);
    p_add(x :> nat(3), y :> nat(2), mk; z, mk);
};
```

3 Implementation Notes

3.1 ProofChecker source code

The dependent type system is formally defined in Figure 1. Its implementation is given below. The main difference is that in the current implementation, type variables are allowed (even if no unification is performed on them).

$$\frac{x\colon\tau \in \Gamma;\Omega}{\Gamma;\Omega \vdash x\colon\tau} \qquad \text{(T.ENV)}$$

$$\frac{}{\Gamma;\Omega \vdash \bar{q}\colon \mathbf{nat}(\mathbf{s}^q(\mathbf{0}))} \qquad \text{(T.NUM)}$$

$$\frac{\vdash_{\mathcal{E}} n = m}{\Gamma;\Omega \vdash * : n = m} \qquad \text{(T.EQUAL)}$$

$$\frac{\Gamma;\Omega \vdash e\colon \tau[n/i] \qquad \Gamma;\Omega \vdash e\colon n = m}{\Gamma;\Omega \vdash e\colon \tau[m/i]} \qquad \text{(T.SUBST-I)}$$

$$\frac{\Gamma;\Omega \vdash s \rhd \Omega[n/i] \qquad \Gamma;\Omega \vdash e\colon n = m}{\Gamma;\Omega \vdash s \rhd \Omega[m/i]} \qquad \text{(T.SUBST-II)}$$

$$\frac{}{\Gamma;\Omega \vdash \varepsilon \rhd \Omega} \qquad \text{(T.EMPTY)}$$

$$\frac{\Gamma;\Omega \vdash e\colon\tau \qquad \Gamma, y\colon\tau;\Omega \vdash s \rhd \Omega'}{\Gamma;\Omega \vdash \mathbf{cst}\ \ y = e;\ \ s\ \rhd \Omega'} \qquad \text{(T.CST)}$$

$$\frac{\Gamma;\Omega \vdash e\colon\tau \qquad \Gamma;\Omega, y\colon\tau \vdash s \rhd \Omega', y\colon\tau'}{\Gamma;\Omega \vdash \mathbf{var}\ \ y := e;\ \ s\ \rhd \Omega'} \qquad \text{(T.VAR)}$$

$$\frac{\Gamma;\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\tau} \qquad \Gamma;\Omega,\vec{x}:\vec{\tau} \vdash s' \rhd \Omega', \vec{x}:\vec{\tau}'}{\Gamma;\Omega,\vec{x}:\vec{\sigma} \vdash \{s\}_{\vec{x}}; s' \rhd \Omega', \vec{x}:\vec{\tau}'} \qquad \text{(T.BLOCK)}$$

$$\frac{\Gamma;\Omega, y\colon \mathbf{nat}(\mathbf{s}(n)) \vdash s \rhd \Omega', y\colon\tau}{\Gamma;\Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{inc}(y); s \rhd \Omega', y\colon\tau} \qquad \text{(T.INC)}$$

$$\frac{\Gamma;\Omega, y\colon \mathbf{nat}(\mathbf{p}(n)) \vdash s \rhd \Omega', y\colon\tau}{\Gamma;\Omega, y\colon \mathbf{nat}(n) \vdash \mathbf{dec}(y); s \rhd \Omega', y\colon\tau} \qquad \text{(T.DEC)}$$

$$\frac{\Gamma;\Omega, y\colon\sigma \vdash e\colon\tau \qquad \Gamma;\Omega, y\colon\tau \vdash s \rhd \Omega', y\colon\tau'}{\Gamma;\Omega, y\colon\sigma \vdash y := e; s \rhd \Omega', y\colon\tau'} \qquad \text{(T.ASSIGN)}$$

$$\frac{\Gamma;\Omega,\vec{x}:\vec{\sigma}[\mathbf{0}/i] \vdash e\colon \mathbf{nat}(n) \quad \Gamma, y\colon\mathbf{nat}(i);\vec{x}:\vec{\sigma} \vdash s \rhd \vec{x}:\vec{\sigma}[\mathbf{s}(i)/i] \quad \Gamma;\Omega,\vec{x}:\vec{\sigma}[n/i] \vdash s' \rhd \Omega',\vec{x}:\vec{\sigma}'}{\Gamma;\Omega,\vec{x}:\vec{\sigma}[\mathbf{0}/i] \vdash \mathbf{for}\ \ y := 0\ \mathbf{until}\ e\ \{s\}_{\vec{x}}; s' \rhd \Omega', \vec{x}:\vec{\sigma}'} \qquad \text{(T.FOR)}^*$$

$$\frac{\vec{z} \neq \emptyset \qquad \Gamma, \vec{y}:\vec{\sigma}; \vec{z}:\vec{\top} \vdash s \rhd \vec{z}:\vec{\tau}[\vec{u}/\vec{\jmath}]}{\Gamma;\Omega \vdash \mathbf{proc}\ (\mathbf{in}\ \vec{y};\mathbf{out}\ \vec{z})\{s\}_{\vec{z}}\colon \mathbf{proc}\ (\{\vec{\imath}\,\}\mathbf{in}\ \vec{\sigma};\{\vec{\jmath}\,\}\mathbf{out}\ \vec{\tau})} \qquad \text{(T.PROC)}^*$$

$$\frac{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p\colon \mathbf{proc}\ (\{\vec{\imath}\,\}\mathbf{in}\ \vec{\tau};\{\vec{\jmath}\,\}\mathbf{out}\ \vec{\sigma}) \quad \Gamma;\Omega,\vec{r}:\vec{\omega} \vdash \vec{e}:\vec{\tau}[\vec{u}/\vec{\imath}] \quad \Gamma;\Omega,\vec{r}:\vec{\sigma}[\vec{u}/\vec{\imath}] \vdash s \rhd \Omega',\vec{r}:\vec{\sigma}'}{\Gamma;\Omega,\vec{r}:\vec{\omega} \vdash p(\vec{e};\vec{r}); s \rhd \Omega', \vec{r}:\vec{\sigma}'} \qquad \text{(T.CALL)}^*$$

*where $\vec{\imath} \notin \mathcal{FV}(\Gamma)$ in (T.PROC), $\vec{\jmath} \notin \mathcal{FV}(\Gamma,\Omega,\Omega',\vec{\sigma}')$ in (T.CALL) and $i \notin \mathcal{FV}(\Gamma)$ in (T.FOR)

Figure 1. Imperative dependent type system

```
(**
 * check that the given sequence represents a correct proof.
 *
 * @params gamma omega (j, omega') reg seq
 * @param gamma the constant proof checking environment.
 * @param omega the variable proof checking environment.
 * @param j the out type existential quantified variables.
 * @param omega' the out type.
 * @param reg the region.
 * @param seq the sequence to check.
 *
 * @return true if and only if seq is a correct proof.
 *)
fun checkSequence gamma omega (j, omega') reg seq =
        let val (iseq, (k, delta)) = seq
        in (* Checks whether the out type is equal to the sequence type annotation *)
                ((envEquals delta omega') andalso (Utils.listEquals k j)
                 andalso (checkInSequence gamma omega (k, delta) reg iseq)) (* Checks the internal sequence *)
                orelse typeFailureSeq gamma omega (j, omega') seq reg "ProofChecker.checkSequence: out type and annotation mismatch"
```

```
        end

and checkInSequence gamma omega (j, omega') reg = fn
        (* Rule T.EMPTY *)
          Empty (subst) =>
                (* We apply the substitution to build the instantiated type *)
                let val out_subst = substituteAllList subst omega'
                        (* We restrict omega to omega' *)
                        val delta = List.filter (fn (x,t) => ListAssoc.memberKey x omega') omega
                in
                        ((Utils.listEquals j (List.map #1 subst)) (* No alpha-equivalence ! Variables must be identical *)
                        andalso (envEquals out_subst delta)) (* The instantiated type must be equal to omega *)
                        orelse typeFailureInSeq gamma omega (j, omega') reg
                                        "ProofChecker.checkSequence: T.EMPTY"
                end

        (* Rule T.CST *)
        | Cst (id, typ, exp, seq) =>
                ((checkExp gamma omega typ reg exp)
                andalso (checkSequence (gamma +! (id, typ)) omega (j, omega') reg seq))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.CST"

        (* Rule T.VAR *)
        | Var (id, typ, exp, seq) =>
                ((checkExp gamma omega typ reg exp)
                andalso (checkSequence gamma (omega +! (id, typ)) (j, omega') reg seq))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.VAR "

        (* Rule T.BLOCK *)
        | Comm (Block (seq1, (k, x_tau_list)), seq2) =>
                (case split omega (dom x_tau_list) of (* We try to split omega according to the out list *)
                  NONE => false (* Failure *)
                | SOME (x_sigma_list, omega'') =>
                (checkSequence gamma x_sigma_list (k, x_tau_list) reg seq1)
                andalso (List.all (fn id => not (Utils.listMember id ((fv_formulas (img gamma)) (* k not-in FV(G,W) *)
                                                                                                                @
(fv_formulas (img omega))))
                                                                        andalso ((Utils.listMember id j) (* k in j or k not-in
FV(W')*)
                                                                        orelse (not (Utils.listMember id (fv_formulas (img
omega'))))))
                                                k)
                andalso (checkSequence gamma (omega'' ++ x_tau_list) (j, omega') reg seq2))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.BLOCK"

        (* Rule T.INC *)
        | Comm (Inc (id, typ), seq) =>
                (case getVarType id omega of
                  SOME (TNat (t)) =>
                        (typ = TNat (t)) andalso
                        checkSequence gamma (omega +! (id, TNat (s(t)))) (j, omega') reg seq
                | _ => false)
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.INC"

        (* Rule T.DEC *)
        | Comm (Dec (id, typ), seq) =>
                (case getVarType id omega of
                  SOME (TNat (t)) =>
                        (typ = TNat (t)) andalso
                        checkSequence gamma (omega +! (id, TNat (p(t)))) (j, omega') reg seq
                | _ => false)
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.DEC"

        (* Rule T.ASSIGN *)
        | Comm (Affect (id, typ, exp), seq) =>
                (case getVarType id omega of
                  NONE => false
                | _ => (checkExp gamma omega typ reg exp)
                                andalso (checkSequence gamma (omega +! (id, typ)) (j, omega') reg seq))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.ASSIGN"

        (* Rule T.FOR *)
        | Comm (For (id, lid, exp, typ, (seq1, (k, sigma_i))), seq2) =>
                (case (typ, split omega (dom sigma_i)) of (* We try to split omega according to the out list *)
                  (TNat (n), SOME (sigma_0, omega'')) =>
                        (envEquals sigma_0 (substituteAll lid FZero sigma_i)) (* sigma[0] must be obtained *)
                        andalso (checkExp gamma omega typ reg exp)
                        andalso (List.length k = 0) (* k must be empty: no existential quantification in recursion *)
                        andalso (checkSequence (gamma +! (id, TNat (FId (lid)))) sigma_i (* sigma[i] |- sigma[s(i)] *)
                                                ([], (substituteAll lid (s(FId (lid))) sigma_i)) reg seq1)
                        andalso (checkSequence gamma (omega'' ++ (substituteAll lid n sigma_i)) (* sigma[n] *)
                                                (j, omega') reg seq2)
                        andalso not (Utils.listMember lid (fv_formulas (img gamma))) (* i not_in FV(G) *)
                | _ => false)
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.FOR"
```

```
        (* Rule T.CALL *)
        | Comm (ProcCall (exp, typ, exp_typ_list, subst, id_typ_list), seq) =>
                ((checkExp gamma omega typ reg exp)
                andalso List.all (fn (e, t) => checkExp gamma omega t reg e) exp_typ_list
                (* We split both omega and omega' to check type equalities *)
                andalso
                        (case (typ, split omega (dom id_typ_list)) of
                          (TProc (i, tau_list, k, sigma_list), SOME (_, omega'')) =>
                                (Utils.listEquals i (List.map #1 subst)) (* No alpha-equivalence ! Variables must be identical *)
                                andalso
                                (let val tau_subst = List.map (substituteList subst) tau_list (* tau[n/i] *)
                                         val typ_list = List.map #2 exp_typ_list
                                 in (* exp_list types = tau[n/i] *)
                                        (ListPair.all (fn (t1, t2) => alphaEqual(t1, t2)) (tau_subst, typ_list))
                                        orelse typeFailureInSeq (List.map (fn t => (("_", 0), t)) tau_subst)
                                                        (List.map (fn t => (("_", 0), t)) typ_list)
                                                        (j, omega') reg "ProofChecker.checkSequence: T.CALL !!!"
                                 end)
                                andalso (List.length id_typ_list = List.length sigma_list)
                                andalso (ListPair.all (fn (t1, t2) => (substituteList subst t1) = t2)
                                                        (sigma_list, List.map #2 id_typ_list)) (* id_list types = sigma[n/i] *)
                                andalso (checkSequence gamma (omega'' ++ id_typ_list) (j, omega') reg seq)
                                andalso (List.all (fn id => not (Utils.listMember id ((fv_formulas (img gamma)) (* k not-in FV(G,W) *)

         @ (fv_formulas (img omega))))
                                                                                        andalso ((Utils.listMember id j) (* k in j or
k not-in FV(W') *)
                                                                                        orelse (not (Utils.listMember id (fv_formulas
(img omega'))))))
                                                            k)
                        | _ => false))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.CALL"

        (* Rule T.SUBST-II *)
        | SSubst (seq, delta_i, lid, exp, typ) =>
                ((checkExp gamma omega typ reg exp)
                andalso (case typ of
                                  TEqual (n, m) =>
                                  (checkSequence gamma omega (j, (substituteAll lid n delta_i)) reg seq) (* delta[n/i] *)
                                  andalso (envEquals (substituteAll lid m delta_i) omega') (* delta[m/i] *)
                                | _ => false))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.SUBST-II"

        (* Rule T.LABEL *)
        | Comm (Label (id, typ, (seq1, (k, id_typ_list))), seq2) =>
                (case (typ, split omega (dom id_typ_list)) of
                  (TProc (k', sigma, [], [TBot]), SOME (x_tau, omega'')) =>
                  (Utils.listEquals sigma (img id_typ_list))
                  andalso (Utils.listEquals k k') (* No alpha-equivalence ! Variables must be identical *)
                  andalso (checkSequence (gamma +! (id, typ)) x_tau (k, id_typ_list) reg seq1)
                  andalso (checkSequence gamma (omega'' ++ id_typ_list) (j, omega') reg seq2)
                | _ => false)
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.LABEL"

        (* Rule T.JUMP *)
        | Comm (Jump (exp, typ, exp_typ_list, subst, id_typ_list), seq) =>
                ((checkExp gamma omega typ reg exp)
                andalso List.all (fn (e, t) => checkExp gamma omega t reg e) exp_typ_list
                andalso (case (typ, split omega (dom id_typ_list)) of
                                  (TProc (k, sigma, [], [TBot]), SOME (_, omega'')) =>
                                  (Utils.listEquals k (List.map #1 subst)) (* No alpha-equivalence ! Variables must be identical *)
                                  andalso
                                        (let val sigma_subst = List.map (substituteList subst) sigma (* sigma[n/i] *)
                                                 val typ_list = List.map #2 exp_typ_list
                                        in (* We check whether exp_list types = sigma[n/i] *)
                                                 (ListPair.all (fn (t1, t2) => alphaEqual(t1, t2)) (sigma_subst, typ_list))
                                                 orelse typeFailureInSeq (List.map (fn t => (("_", 0), t)) sigma_subst)
                                                                         (List.map (fn t => (("_", 0), t)) typ_list)
                                                                         (j, omega') reg "ProofChecker.checkSequence: T.JUMP !!!"
                                        end)
                                  andalso (checkSequence gamma (omega'' ++ id_typ_list) (j, omega') reg seq)
                                | _ => false))
                orelse typeFailureInSeq gamma omega (j, omega') reg
                                "ProofChecker.checkSequence: T.JUMP"

        | Comm (CommRegion (c, reg), seq) => checkInSequence gamma omega (j, omega') reg (Comm (c, seq))
        | SeqRegion (seq, reg) => checkInSequence gamma omega (j, omega') reg seq

(**
 * check that the given expression represents a correct proof.
 *
 * @params gamma omega typ reg (expression, tau)
 * @param gamma the constant proof checking environment.
 * @param omega the variable proof checking environment.
 * @param typ the requiered type.
 * @param reg the region.
 * @param expression the expression to check.
 * @param tau the given type.
```

```
 *
 * @return true if and only if expression is a correct proof of formula tau.
 *)
and checkExp gamma omega typ reg (expression, tau) =
        (tau = typ) andalso
        case expression of
        (* Rule T.ENV *)
          Id (id) =>
          (case getVarType id (gamma ++ omega) of
            NONE => false
          | SOME t => t = typ)
          orelse typeFailureExp gamma omega typ expression reg
                                "ProofChecker.checkExp: T.ENV"

        | Val (v) => checkValue gamma omega typ reg v

        (* Rule T.SUBST-I *)
        | ESubst (exp1, typ1, lid, exp2, typ2) =>
        ((checkExp gamma omega typ2 reg exp2)
        andalso (case typ2 of
                          TEqual (n, m) =>
                          (checkExp gamma omega (substitute lid n typ1) reg exp1)
                          andalso (typ = (substitute lid m typ1))
                        | _ => false))
        orelse typeFailureExp gamma omega typ expression reg
                        "ProofChecker.checkExp: T.SUBST-I"

        (* Lemma *)
        | Lemma (typlist, typ') =>
        (let val hyp = img (gamma ++ omega)
        in
                (typ = typ') andalso
                (List.all (fn x => Utils.listMember x hyp) typlist)
        end)
        orelse typeFailureExp gamma omega typ expression reg
                        "ProofChecker.checkExp: Lemma"

        | ExpRegion (exp, reg) => checkExp gamma omega typ reg (exp, tau)

(**
 * check that the given value represents a correct proof.
 *
 * @params v reg
 * @param v the value.
 *
 * @return true if and only if p is a correct proof.
 *)
and checkValue gamma omega typ reg (value, tau) =
        (tau = typ) andalso
        case value of
        (* Rule T.STAR *)
          Star =>
        (case typ of TEqual (n, m) => true
                                | _ => false)
        orelse typeFailureExp gamma omega typ (Val (value, tau)) reg
                        "ProofChecker.checkValue: T.STAR"

        (* Rule T.NUM *)
        | Int (n) =>
        (typ = (TNat (mkNat n)))
        orelse typeFailureExp gamma omega typ (Val (value, tau)) reg
                        "ProofChecker.checkValue: T.NUM"

        (* Rule T.PROC *)
        | Proc (i, in_list, j, out_list, (seq, (j', typid_list))) =>
        (let val sigmalist = img in_list
                 val taulist = img out_list
         in
                 (typ = (TProc (i, sigmalist, j, taulist)))
                 andalso (Utils.listEquals j j') (* No alpha-equivalence ! Variables must be identical *)
                 andalso (envEquals out_list typid_list)
                 andalso (checkSequence (in_list ++ gamma) (List.map (fn (x,t) => (x, top)) typid_list)
                                        (j', typid_list) reg seq)
                 andalso (List.all (fn id => not (Utils.listMember id (fv_formulas (img gamma)))) i) (* i not-in FV(G) *)
         end)
        orelse typeFailureExp gamma omega typ (Val (value, tau)) reg
                                  "ProofChecker.checkValue: T.PROC"

        | ValRegion (v, reg) => checkValue gamma omega typ reg (v, tau)

(**
 * check that the given program represents a correct proof.
 *
 * @params p
 * @param p the program.
 *
 * @return true if and only if p is a correct proof.
 *)
fun checkClosedProg (p : program) =
        case p of
          (SeqRegion (seq, reg), seqType) => checkSequence empty empty seqType reg p
        | _ => raise Fail "ProofChecker.checkClosedProg: not a SeqRegion"
```

4 Source code documentation (Smldoc)

First-Order LoopW Type Inference

Inner Signature summary
signature BASE_PRETTY_PRINTER The First-Order LoopW Type Inference base pretty-printer.
signature COMPILER The First-Order LoopW Type Inference compiler.
signature CORE_AST_UTILS Utility function for the core syntax.
signature CORE_PRETTY_PRINTER The First-Order LoopW Type Inference core syntax pretty-printer.
signature LIST_ASSOC Utility functions that deals with association lists.
signature PRETTY_PRINTER_KEYWORDS The First-Order LoopW Type Inference pretty-printer keywords for abstract syntax.
signature PROOF_DERIV_PRETTY_PRINTER The First-Order LoopW Type Inference proof derivation pretty-printer.
signature PROOF_ENVIRONMENT The First-Order LoopW Type Inference proof environment abstract type.
signature PROOF_INFER The First-Order LoopW Type Inference proof inference.
signature PROOF_PRETTY_PRINTER The First-Order LoopW Type Inference proof syntax pretty-printer.
signature TERMS The First-Order LoopW Type Inference formula/term substitution and alpha-equivalence.
signature TERM_MATCH_UNIF The First-Order LoopW Type Inference formula/term matching and unification.
signature UTILS Utility functions that deals with lists.

Inner Structure summary
structure BasePrettyPrinter The First-Order LoopW Type Inference base pretty-printer.
structure Compiler The First-Order LoopW Type Inference compiler.
structure CoreAst The First-Order LoopW Type Inference core abstract syntax.
structure CoreAstUtils Utility function for the core syntax.

`structure` CorePrettyPrinter The First-Order LoopW Type Inference core syntax pretty-printer.
`structure` Error The First-Order LoopW Type Inference syntax error module.
`structure` ListAssoc Utility functions that deals with association lists.
`structure` Main The First-Order LoopW Type Inference main program.
`structure` Parser
`structure` PrettyPrinterKeywords The First-Order LoopW Type Inference pretty-printer keywords for abstract syntax.
`structure` ProofAst The First-Order LoopW Type Inference proof abstract syntax.
`structure` ProofDerivPrettyPrinter The First-Order LoopW Type Inference proof derivation pretty-printer.
`structure` ProofEnvironment The First-Order LoopW Type Inference proof environment abstract type.
`structure` ProofInfer The First-Order LoopW Type Inference proof inference.
`structure` ProofPrettyPrinter The First-Order LoopW Type Inference proof syntax pretty-printer.
`structure` TermMatchUnif The First-Order LoopW Type Inference formula/term matching and unification.
`structure` Terms The First-Order LoopW Type Inference formula/term substitution and alpha-equivalence.
`structure` Utils Utility functions that deals with lists.

Inner Functor summary
`functor` SyntaxLexFun

First-Order LoopW

Inner Signature summary

signature BASE_PRETTY_PRINTER The First-Order LoopW Proof Checker base pretty-printer.
signature COMPILER The First-Order LoopW Proof Checker compiler.
signature FUN_PRETTY_PRINTER The First-Order LoopW Proof Checker functional pretty-printer.
signature FUN_TRANSLATION The First-Order LoopW Proof Checker translator into functional syntax.
signature LIST_ASSOC Utility functions that deals with association lists.
signature PRETTY_PRINTER_KEYWORDS The First-Order LoopW Proof Checker pretty-printer keywords for abstract syntax.
signature PROOF_AST_UTILS Utility function for the proof syntax.
signature PROOF_CHECKER The First-Order LoopW Proof Checker proof checking.
signature PROOF_ENVIRONMENT The First-Order LoopW Proof Checker proof environment abstract type.
signature PROOF_PRETTY_PRINTER The First-Order LoopW Proof Checker proof syntax pretty-printer.
signature TERMS The First-Order LoopW Proof Checker formula/term substitution and alpha-equivalence.
signature UTILS Utility functions that deals with lists.

Inner Structure summary

structure BasePrettyPrinter The First-Order LoopW Proof Checker base pretty-printer.
structure Compiler The First-Order LoopW Proof Checker compiler.
structure Error The First-Order LoopW Proof Checker syntax error module.
structure FunAst The First-Order LoopW Proof Checker functional programs abstract syntax.
structure FunPrettyPrinter The First-Order LoopW Proof Checker functional pretty-printer.

`structure` FunTranslation The First-Order LoopW Proof Checker translator into functional syntax.
`structure` FunTypeChecker
`structure` ListAssoc Utility functions that deals with association lists.
`structure` Main The First-Order LoopW Proof Checker main program.
`structure` Parser
`structure` PrettyPrinterKeywords The First-Order LoopW Proof Checker pretty-printer keywords for abstract syntax.
`structure` ProofAst The First-Order LoopW Proof Checker proof abstract syntax.
`structure` ProofAstUtils Utility function for the proof syntax.
`structure` ProofChecker The First-Order LoopW Proof Checker proof checking.
`structure` ProofEnvironment The First-Order LoopW Proof Checker proof environment abstract type.
`structure` ProofPrettyPrinter The First-Order LoopW Proof Checker proof syntax pretty-printer.
`structure` Terms The First-Order LoopW Proof Checker formula/term substitution and alpha-equivalence.
`structure` Utils Utility functions that deals with lists.

Inner Functor summary
`functor` SyntaxLexFun