PROSPER

Proof and Specification Assisted Design Environments
ESPRIT LTR Project 26241

IL Concrete Syntax
Version 1.0 β

PUBLIC

27 May 1999

Mike Gordon, Michael Norrish
University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street, Cambridge, UK, CB2 3QG

IL Concrete Syntax

Mike Gordon, Michael Norrish

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England

27 May 1999

Abstract

The semantic framework underlying the Intermediate Language (IL) is described in Deliverable D2.1c. This note gives a *preliminary concrete syntax* for the IL semantic operators. The aim of this syntax is to provide a minimal starting point for the translation of HDL into IL. It is expected that the details given here will evolve when the actual translation of HDL to IL is undertaken.

Contents

1 Table of showing syntax of IL in HOL

D2.1c Notation	**Preliminary HOL concrete syntax**

`module` <*name*>(<*parameter*>,...,<*parameter*>)
<*body*>
`end`
	<*identifier*> `=`
	`Module`
	`(`<*identifier*>`,` ... `,`<*identifier*>`).`
	<*term*>

`local` <*variable*> ⋯ <*variable*>. <*body*>	`Local` <*identifier*> ⋯ <*identifier*>. <*term*>
<*item*>`;` ⋯ `;` <*item*>	<*term*> `&` ⋯ `&` <*term*>
`init` <*variable*> = <*expression*>	`init` <*identifier*> = <*term*>
<*variable*> = <*expression*>	<*identifier*> = <*term*>
`next` <*variable*>	`next` <*identifier*>
<*variable*> `:=` <*expression*>	<*identifier*> `:=` <*term*>
<*guard*> `->` <*variable*> `:=` <*expression*>	<*term*> `-->` `(`<*identifier*>`,`<*term*>`)`
`rise` <*variable*>	`rise` <*term*>
`fall` <*variable*>	`fall` <*term*>
`change` <*variable*>	`change` <*term*>
`change(`<*variable*>`,` ... `,`<*variable*>`)`	`change_any[`<*term*>`;` ... `;`<*term*>`]`
<*constant*>	`lift` <*constant*>
<*unary*> <*expression*>	`lift1` <*unary*> <*term*>
<*expression*> <*binary*> <*expression*>	`lift2 $`<*binary*> <*term*> <*term*>

2 Examples

The following example is from Section 4.4 of D2.1c:

module Del (i,o)
 $o := i$
end

module Del2 (i,o)
 local x.
 Del(i,x);
 Del(x,o)
end

module Del4 (i,o)
 local x.
 Del2(i,x);
 Del2(x,o)
end

Using the proposed HOL concrete syntax this is:

```
Del = (Module (inp, out).
         init out = v0 &
         out := inp)
&
Del2 = (Module (inp, out).
         Local y. Del(inp, y) & Del(y, out))
&
Del4 = (Module (inp, out).
         Local x1 x2.
           Del2(inp, x1)    &
           x2 = lift1 f x1  &
           Del2(x2, out))
```

A positive edge triggered Dtype register initilised to store F (i.e. falsity or 0) is defined in Section 4.7 of D2.1c by:

module Dreg $(clk,d,q,\overline{q})$
 init $q = \mathsf{F}$;
 rise clk -> $q := d$;
 $\overline{q} = \neg q$
end

Using the HOL concrete syntax this is:

```
Dreg = Module (clk, d, q, qbar).
        init q = F          &
        rise clk --> (q, d) &
        qbar = neg q
```

where `neg` is defined to be `lift1 $~` (i.e. $\forall t.$ `neg q` t = `~(q` t`)`)

3 Shallow embedding in HOL

The following definitions of the operators in Section 1 may be loaded into `Hol98 [Athabasca 4]`.

```
(* Load infrastructure *)
app (fn s => (print ("Loading "^s^"\n"); load s))
    ["rich_listTheory",
     "bossLib"
    ];

val Define = bossLib.Define;

(* IL assignments *)
val next_def        = Define `next V t = V (t + 1)`;
val next_assignment = Define `$:= V E = (!t:num. V(t+1) = E t)`;
set_fixity ":=" (Infix 10);
set_MLname ":=_def" "next_assignment";

val event_guard =
  Define `$--> P (V,E) = (!t:num. V(t+1) = (P t => E t | V t))`;
set_fixity "-->" (Infix 10);
set_MLname "-->_def" "event_guard";

(* clock edge events *)
val rise_def       = Define `rise clk t = ~(clk t) /\ clk (t + 1)`;
val fall_def       = Define `fall clk t = clk t /\ ~clk (t + 1)`;
val change_def     = Define `change V t = ~(V t = V (t + 1))`;
val change_any_def = Define `change_any Vs t = SOME_EL (\v. change v t) Vs`;


(* sequential composition *)
val seqcomp_def = Define `& p q = p /\ q`;
set_fixity "&" (Infix 9);
set_MLname "&_def" "seqcomp_def";

(* initialisation *)
val init_def = Define `init V = V 0`;

(* modules *)
val Module_def = Define `$Module = (I:('a -> bool) -> ('a -> bool))`;
set_fixity "Module" Binder;

(* local *)

val Local_def = Define `$Local = $?`;
set_fixity "Local" Binder;

(* lifting *)
val lift_def  = Define `lift (x:'a) = \t:num. x`;
val lift1_def = Define `lift1 f V = f o V`;
val lift2_def = Define `lift2 f V1 V2 = \t:num. f (V1 t) (V2 t)`;
val lift3_def = Define `lift3 f v1 v2 v3 = \t:num. f (v1 t) (v2 t) (v3 t)`;
val neg_def   = Define `neg = lift1 $~`;
```

4 Example simplification using HOL

Using the definition of `Del4` above HOL can automatically derive:

```
Del4(inp,out) =
 ?x1.
  (?y. ((y 0 = v0) /\ (!t. y (t + 1) = inp t))    /\
       (x1 0 = v0)                                /\
       (!t. x1 (t + 1) = y t))
  /\
  (?y. ((y 0 = v0) /\ (!t. y (t + 1) = f (x1 t))) /\
       (out 0 = v0)                               /\
       (!t. out (t + 1) = y t))
```

Using the definition of `Dreg` HOL can automatically derive:

```
Dreg(clk,d,q,qbar) =
 ~(q 0)                                                   /\
 (!t. q (t + 1) = ((~(clk t) /\ clk (t + 1)) => (d t) | (q t))) /\
 (qbar = ~ o q)
```

where `o` in the last line is functional composition.

5 Complete ML code

```
app (fn s => (print ("Loading "^s^"\n"); load s))
    ["rich_listTheory",
     "bossLib",
     "tautLib",
     "simpLib",
     "HOLSimps",
     "Q"] ;

val Define = bossLib.Define;

(* IL assignments *)
val next_def = Define `next V t = V (t + 1)`;
val next_assignment = Define `$:= V E = (!t:num. V(t+1) = E t)`;
set_fixity ":=" (Infix 10);
set_MLname ":=_def" "next_assignment";

val event_guard =
  Define `$--> P (V,E) = (!t:num. V(t+1) = (P t => E t | V t))`;
set_fixity "-->" (Infix 10);
set_MLname "-->_def" "event_guard";

(* clock edge events *)
val rise_def = Define `rise clk t = ~(clk t) /\ clk (t + 1)`;
val fall_def = Define `fall clk t = clk t /\ ~clk (t + 1)`;
val change_def = Define `change V t = ~(V t = V (t + 1))`;
val change_any_def = Define `change_any Vs t = SOME_EL (\v. change v t) Vs`;

(* initialisation *)
val init_def = Define `init V = V 0`;

(* "sequential composition" *)
val seqcomp_def = Define `& p q = p /\ q`;
set_fixity "&" (Infix 9);
set_MLname "&_def" "seqcomp_def";

(* modules *)
val Module_def = Define `$Module = (I:('a -> bool) -> ('a -> bool))`;
set_fixity "Module" Binder;

(* local *)
val Local_def = Define `$Local = $?`;
set_fixity "Local" Binder;
```

```
(* lifting *)
val lift_def = Define `lift (x:'a) = \t:num. x`;
val lift1_def = Define `lift1 f V = f o V`;
val lift2_def = Define `lift2 f V1 V2 = \t:num. f (V1 t) (V2 t)`;
val lift3_def = Define `lift3 f v1 v2 v3 = \t:num. f (v1 t) (v2 t) (v3 t)`;
val neg_def = Define `neg = lift1 $~`;

val Del2 =
  ``Del2 = Module (inp, out).
            Local Del y.
            (Del = Module (inp, out). out := inp) &
            Del(inp, y) &
            Del(y, out)``;

val Del2_flattened =
 (simpLib.SIMP_CONV
    HOLSimps.hol_ss
    [Local_def, Module_def, seqcomp_def, next_assignment]
   THENC DEPTH_CONV AND_FORALL_CONV)
 Del2;

val init_comb_example =
  ``Del = (Module (inp, out).
            init out = v0 &
            out := inp) &
    Del2 = (Module (inp, out).
            Local y. Del(inp, y) & Del(y, out)) &
    Del4 = (Module (inp, out).
            Local x1 x2.
              Del2(inp, x1)   &
              x2 = lift1 f x1 &
              Del2(x2, out))``;

val CONJ_ss =
 simpLib.SIMPSET
   {ac     = [],
    convs  = [],
    dprocs = [],
    filter = NONE,
    rewrs  = [],
    congs  = [GEN_ALL (tautLib.TAUT_PROVE
                        ``(P ==> (Q = Q')) ==> (Q' ==> (P = P'))
                          ==>
                          ((P /\ Q) = (P' /\ Q'))``)]
   };

val ncase_intro =
 prove
  (``!f v g. (f 0 = v) /\ (!n. f (n + 1) = g n) = (f = num_case v g)``,
   REPEAT GEN_TAC
    THEN CONV_TAC (RAND_CONV FUN_EQ_CONV)
    THEN EQ_TAC
    THEN REPEAT STRIP_TAC
    THENL
     [STRUCT_CASES_TAC (Q.SPEC `n` arithmeticTheory.num_CASES)
       THEN simpLib.ASM_SIMP_TAC HOLSimps.hol_ss [arithmeticTheory.num_case_def]
       THEN simpLib.ASM_SIMP_TAC HOLSimps.hol_ss [arithmeticTheory.ADD1],
      simpLib.ASM_SIMP_TAC HOLSimps.hol_ss [arithmeticTheory.num_case_def],
      simpLib.ASM_SIMP_TAC HOLSimps.hol_ss [GSYM arithmeticTheory.ADD1,
                                             arithmeticTheory.num_case_def]
     ]);

val ++ = simpLib.++;
infix ++;

val flattened_comb_example =
  simpLib.SIMP_CONV
   (HOLSimps.hol_ss ++ CONJ_ss)
   [Local_def, Module_def, seqcomp_def, next_assignment, init_def, lift1_def]
   init_comb_example;

val really_flat_comb_example =
  simpLib.SIMP_CONV
   (HOLSimps.hol_ss ++ CONJ_ss)
   [Local_def, Module_def, seqcomp_def, next_assignment,
    init_def, lift1_def, ncase_intro]
   init_comb_example;

val dreg_example =
  ``Dreg = Module (clk, d, q, qbar). init q = F          &
                                     rise clk --> (q, d) &
                                     qbar = neg q``;
val dreg_flattened =
  simpLib.SIMP_CONV
   HOLSimps.hol_ss
   [init_def, neg_def, lift1_def, event_guard, Module_def, seqcomp_def, rise_def]
   dreg_example;
```