

Spin2Core translation proposal

Here are some guidelines for translating a spin program to core program:

Constants

The Spin language supports two different ways to define constants:

1. C-style #define
2. Using mtype

Table 1 - Constants

Cons. name	C-equivalent	Typical Range	Core Equivalent
#define name value	#define name value	number	CONST name: value;
#define name string	#define name string	string	DEFINE name: string;
mtype = {id1,...,id256};	Enum mtype {id1,...,id256};	Up to 256 different identifiers	TYPE mtype: ENUM {id1,...,id255};

Data Types

In Spin language there are five basic types. Here is the table of the data types, their C-equivalent and their Core equivalent:

Table 2 - Data Types

Type name	C-equivalent	Typical Range	Core Equivalent
bit or bool	bit-field	0..1	boolean
byte	unsigned char	0..255	1.integer
short	Short	$-2^{15} - 1 .. 2^{15} - 1$	2.TYPE name: range..range; 3.module
int	Int	$-2^{31} - 1 .. 2^{31} - 1$	integer

Expressions

The spin language supports the following expressions:

Table 3 - Expressions

Exp. name	Description	Core Equivalent
+	Plus	+
-	Minus	-
*	Times	*
/	Divide	/
%	Modulo	%,mod
>	greater than	>
>=	greater equal	>=

<	less than	<
<=	less equal	<=
==	Equal	=
!=	not equal	!=
!	Not	!
&	bit wise and	Module bit_and
	Or	∨
&&	And	∧
	bit wise or	Module bit_or
~	bit wise not	Module bit_not
>>	shift right	Module shift_right
<<	shift left	Module shift_left
^	exclusive or	$(x \wedge (!y)) \vee ((!x) \wedge y)$
++	plus-plus (suffix only)	$x' := x + 1;$
--	minus-minus (suffix only)	$x' := x - 1;$
-> :	operator ?:	if expression

Problems:

The bit wise operations are not supported by core, and will require either extending the core language or designing a module to handle the operations.

Channels

A special type in the Spin language.

Syntax: `chan name = [const] of {typename1, typename2, typename3,...}`

The channel has a buffer of messages in which new message can be put if the buffer is not full and messages can be extracted if the buffer is not empty.

The “send” operation: `name!msg`

The message “msg” must be of type {typename1, typename2, typename3,...} and the buffer must be non-full.

The “receive” operation: `name?msg`

The message “msg” must be of type {typename1, typename2, typename3,...} and the buffer must be non-empty.

Translation suggestion:

HOLD_PREVIOUS

```
MODULE CHAN (msg: integer) {
  VAR
    num_of_msg: integer      INITVAL 0;
    buff:       ARRAY [5] OF integer INITVAL -1;
    get_ok:     boolean      INITVAL  true;
    put_ok:     boolean      INITVAL  false;
    move_buff:  boolean      INITVAL  false;

    COJOIN: ((num_of_msg <= 5) /\ (num_of_msg >= 0));

  TRANS get:
    enable:     (num_of_msg < 5) /\ get_ok;
    assign:    buff[num_of_msg]' := msg;
              num_of_msg' := num_of_msg + 1;
              put_ok' := true;

  TRANS full:
    enable:     (num_of_msg = 5);
    assign:    get_ok' := false;

  TRANS empty:
    enable:     (num_of_msg = 0);
    assign:    put_ok' := false;

  TRANS put:
    enable:     ((!move_buff) /\ (put_ok));
    assign:    put_ok' := false;
              move_buff' := true;
              num_of_msg' := num_of_msg - 1;
              msg' := (buff[0]);

  TRANS move:
    enable:     move_buff;
    assign:    buff[0]' := (buff[1]);
              buff[1]' := (buff[2]);
              buff[2]' := (buff[3]);
              buff[3]' := (buff[4]);
              buff[4]' := -1;
              move_buff' := false;
              get_ok' := true;
              put_ok' := true;
}

MODULE SENDER (h: integer) {
  VAR
    readys: boolean INITVAL false;

    COJOIN : (h < 4);

  TRANS produce:
    enable: !readys;
    assign: readys' := true;

  TRANS send:
    enable: readys;
    assign: readys' := false;
}
}
```

```

MODULE RECEIVER (l: integer) {
  VAR
  vr: integer;
  readyr: boolean INITVAL true;

  TRANS consume:
    enable: !readyr;
    assign: vr' := l;
    relation: (vr' < 5) /\ readyr;

  TRANS recieve:
    enable: readyr;
    assign: l' := {2, 5, 4};
           readyr' := false;
}

MODULE SYSTEM () {
  VAR
  s: integer;
  t: integer;

  (( SENDER(s))(send,get)CHAN(s) || ( CHAN(t))(put,receive) RECEIVER(t) )
}

```

Problem:

1. Channel can be passed as a parameter to other processes. Implementing it as MOUDLE will make it difficult
2. Channel can be also a rendezvous.
3. Channel has expression indicating the state of the channel: len(), empty(), nempty(), nfull(), full().
4. Channel can be exclusive (belongs to one process only).

Arrays

An array in Spin is declared:

```
typename  name[const] =initializer;
```

The Core equivalent will be:

```
name:      ARRAY[const] OF typename INITVAL initializer;
```

Structs

The Spin supports C-like structs through the typedef definitions:

```
typedef name {
  attributes list
}
```

Processes

The basic elements of the Spin language:

```
proctype name (parameters list) { statements }
```

translation suggestion: each process will be translated to MODULE. The module will keep PC variable, since the Spin supports flow control, and a process considered blocked if current statement isn't executable.