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>
<thead>
<tr>
<th>Cons. name</th>
<th>C-equivalent</th>
<th>Typical Range</th>
<th>Core Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define name value</td>
<td>#define name value</td>
<td>number</td>
<td>CONST name: value;</td>
</tr>
<tr>
<td>#define name string</td>
<td>#define name string</td>
<td>string</td>
<td>DEFINE name: string;</td>
</tr>
<tr>
<td>mtype = {id1,…,id256};</td>
<td>Enum mtype {id1,…,id256};</td>
<td>Up to 256 different identifiers</td>
<td>TYPE mtype: ENUM {id1,…,id255};</td>
</tr>
</tbody>
</table>

**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>
<thead>
<tr>
<th>Type name</th>
<th>C-equivalent</th>
<th>Typical Range</th>
<th>Core Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit or bool</td>
<td>bit-field</td>
<td>0..1</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>unsigned char</td>
<td>0..255</td>
<td>1.integer</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>$-2^{15}$ - 1 .. $2^{15}$ - 1</td>
<td>2.TYPE name: range..range;</td>
</tr>
<tr>
<td>int</td>
<td>Int</td>
<td>$-2^{31}$ - 1 .. $2^{31}$ - 1</td>
<td>3.module integer</td>
</tr>
</tbody>
</table>

**Expressions**

The spin language supports the following expressions:

<table>
<thead>
<tr>
<th>Exp. name</th>
<th>Description</th>
<th>Core Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>Times</td>
<td>*</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>/</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>%,mod</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater equal</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Module</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less equal</td>
<td>&lt;=</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
<td>=</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>!=</td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
<td>!</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit wise and</td>
<td>Module bit_and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>And</td>
<td>\land</td>
</tr>
<tr>
<td>l</td>
<td>bit wise or</td>
<td>Module bit_or</td>
</tr>
<tr>
<td>~</td>
<td>bit wise not</td>
<td>Module bit_not</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Module shift_right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Module shift_left</td>
</tr>
<tr>
<td>^</td>
<td>exclusive or</td>
<td>(x\land\lnot y)\lor (\lnot x\land y)</td>
</tr>
<tr>
<td>++</td>
<td>plus-plus (suffix only)</td>
<td>x' := x + 1;</td>
</tr>
<tr>
<td>--</td>
<td>minus-minus (suffix only)</td>
<td>x' := x - 1;</td>
</tr>
<tr>
<td>-&gt;</td>
<td>operator ?:</td>
<td>if expression</td>
</tr>
</tbody>
</table>

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: \texttt{chan name = [const] of \{typename1, typename2, typename3,\ldots\}}
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: \texttt{name!msg}
The message “msg” must be of type \{typename1, typename2, typename3,\ldots\} and the buffer must be non-full.
The “receive” operation: \texttt{name?msg}
The message “msg” must be of type \{typename1, typename2, typename3,\ldots\} and the buffer must be non-empty.
MODULE CHAN (msg: integer) {
  VAR
    num_of_msg: integer INITVAL 0;
    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
    ready: boolean INITVAL false;
  COJOIN: (h < 4);

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

  TRANS send: 
    enable: ready;
    assign: ready' := false;
}
MODULE RECEIVER (l: integer) {
  VAR
  vr: integer;
  readyr: boolean INITVAL true;

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

  TRANS receive:
    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 MODULE 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.