LLVM-based code generation for B

Conference Paper · September 2014

DOI: 10.1007/978-3-319-15075-8_1

CITATIONS
6

READS
617

4 authors, including:

Richard Bonichon
Nomadic Labs
29 PUBLICATIONS **320** CITATIONS
SEE PROFILE

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS **940** CITATIONS
SEE PROFILE

Thierry Lecomte
ClearSy System Engineering
54 PUBLICATIONS **418** CITATIONS
SEE PROFILE

LLVM-based code generation for B

Richard Bonichon, David Déharbe, Thierry Lecomte, Valério Medeiros Jr*

UFRN, Brazil and Clearsy, France

Abstract. We present b2llvm, a multi-platform code generator for the B-method. The b2llvm code generator currently handles the following elements of the B language: simple data types, imperative instructions and component compositions. In particular, this paper describes a translation for essential implementation constructs of the B language into LLVM source code, implemented into the b2llvm compiler. We use an example-based approach for this description.

1 Introduction

The B-method is a refinement-based software design method [1]. Its language has both abstract constructs, suitable for declarative-like specifications, and imperative constructs, commonly found in programming languages. B development typically starts with a specification, in a so-called *machine*, followed by incremental refinements to an *implementation*, where only imperative-like constructs may be employed [6]. Such an implementation is then translated [5] to source code in a programming language, say C or Ada. The steps in the B-method are verified using certified theorem proving technologies. However the translation to a programming language, and its subsequent compilation to the target platform, do not benefit from the same mathematical rigor. In practice, *redundancy* in the tool chains and execution platforms is employed to increase the level of confidence to the desired levels.

The goal of this work is to contribute a redundancy element, by creating a new open-source machine-code generation tool chain. To achieve this, we base our work on the LLVM compilation framework [8]. LLVM is an active open-source compiler infrastructure used by many compiling toolchains. It provides an intermediate assembly language suitable to the applications of many compiler techniques such as optimization, static analysis, code generation, debugging. We defined a translation from B0 (the subset of the B language that is used to describe imperative programs) to the LLVM intermediate representation, which is implemented in the b2llvm tool[1].

The rest of the paper is organized as follows. Section 2 presents selected aspects of the LLVM intermediate representation language. Next, in section 3,

* The research presented in this paper was partially supported by CNPq projects 308008/2012-0 and 573964/2008-4 (National Institute of Science and Technology for Software Engineer - INES).

[1] The b2llvm project is hosted at https://www.b2llvm.org/b2llvm.

we review some important concepts of the B-method regarding the structure of projects. A user perspective of the code generator is then presented in section 4. In section 5 we present some details of the code generation process through illustrative examples. Also, section 6 discusses verification and validation aspects. We conclude and consider future work in section 7.

2 Target LLVM Subset

The LLVM project defines an intermediate representation language (LLVM IR), as a means to implement different compiler components. Front-ends translate source programming languages to LLVM IR, optimizers and other static analysis tasks may be applied to the IR, and back-ends translate from LLVM IR to target platform assembly languages. LLVM IR is a single-static assignment (SSA) language, i.e., a variable may only be assigned in a single instruction. Figure 1 exemplifies LLVM IR syntax with a simple program together with its equivalent C program.

```
define void @inc(i32* %pi) {
entry:
  %0 = load i32* %pi
  %1 = add i32 %0, 1
  store i32 %1, i32* %pi
  ret void
}
```

```
void inc(int * pi)
{
 *pi += 1;
}
```

Fig. 1. Simple example of a C function and its corresponding LLVM IR function. The first line contains the signature: return type `void`, the name `@inc` and one parameter named `%pi` and typed `i32*`. Next is the body with a single block, labeled `entry`, and temporary variables `%0` and `%1`, created in the conversion to SSA. The block has four instructions: `load`, `add`, `store` and `ret`. For instance, `%1 = add i32 %0, 1` performs an addition (`add`), has result type `i32` and assigns to `%1` the sum of variable `%0` and integer literal `1`.

Figure 2 presents the subset of LLVM IR targeted by the b2llvm code generator. LLVM IR programs are organized into modules, one per translation unit. A module may contain declarations of external entities (functions and constants) and definitions of internal items (functions, variables and constants). Data must be typed and the name and type of external entities must be declared. All names, e.g. non-reserved identifiers, must start with `@`, when they are global, or `%`, when they are local. For instance, `@max = external constant i32` declares `@max` as a 32-bit integer constant and `declare void @inc(i32*)` declares `@inc` as a function with one parameter, namely a pointer to an integer, and a `void` return type.

The type system contains the empty type `void`, a (countable) infinite, number of integer types, one for each possible bit width (e.g., `i8` is the type for 8-bit integers), and type constructors pointer (declared with monadic operator $\cdot$ `*`) and structure (declared with polyadic operator $\{\cdots\}$). For instance `{ i8*, i8,`

$$
\begin{array}{rcl}
\mathit{module} & ::= & \mathit{item}^{+} \\
\mathit{item} & ::= & \mathit{const\_decl} \mid \mathit{function\_decl} \mid \\
 & & \mathit{type\_def} \mid \mathit{const\_def} \mid \mathit{var\_def} \mid \mathit{function\_def} \\
\mathit{const\_decl} & ::= & \mathit{name}\ \texttt{=}\ \texttt{external constant}\ \mathit{type} \\
\mathit{type\_def} & ::= & \mathit{name}\ \texttt{=}\ \texttt{type}\ \mathit{type} \\
\mathit{type} & ::= & \texttt{void} \mid \mathit{itype} \mid \{\ \mathit{type}^{+}\ \} \mid \mathit{type}\texttt{*} \\
\mathit{const\_def} & ::= & \mathit{name}\ \texttt{=}\ \texttt{constant}\ \mathit{type}\ \mathit{iliteral} \\
\mathit{var\_def} & ::= & \mathit{name}\ \texttt{=}\ \texttt{common global}\ \mathit{type}\ \texttt{zeroinitializer} \\
\mathit{function\_decl} & ::= & \texttt{declare}\ \mathit{type}\ \mathit{name}\ \texttt{(}\ \mathit{type}^{+}\ \texttt{)} \\
\mathit{function\_def} & ::= & \texttt{define}\ \mathit{type}\ \mathit{name}\ \texttt{(}\ \mathit{param}^{+}\ \texttt{)}\ \{\ \mathit{block}^{+}\ \} \\
\mathit{param} & ::= & \mathit{type}\ \mathit{name} \\
\mathit{block} & ::= & \mathit{lbl}\ \texttt{:}\ \mathit{inst}^{+} \\
\mathit{inst} & ::= & \mathit{name}\ \texttt{=}\ \texttt{alloca}\ \mathit{type} \\
 & \mid & \mathit{name}\ \texttt{=}\ \langle\ \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{sdiv} \mid \texttt{srem}\ \rangle \mathit{itype}\ \mathit{exp}\ \texttt{,}\ \mathit{exp} \\
 & \mid & \mathit{name}\ \texttt{=}\ \texttt{icmp}\ \langle\ \texttt{eq} \mid \texttt{ne} \mid \texttt{sgt} \mid \texttt{sge} \mid \texttt{slt} \mid \texttt{sle}\ \rangle\ \texttt{i1}\ \mathit{exp}\ \texttt{,}\ \mathit{exp} \\
 & \mid & \mathit{name}\ \texttt{=}\ \texttt{call}\ \mathit{type}\ \texttt{(}\ \mathit{arg}^{+}\ \texttt{)} \\
 & \mid & \mathit{name}\ \texttt{=}\ \texttt{getelementptr}\ \mathit{type}\ \texttt{*}\ \mathit{exp}\texttt{,}\ \mathit{index}\texttt{,}\ \mathit{index} \\
 & \mid & \mathit{name}\ \texttt{=}\ \texttt{load}\ \mathit{type}\ \mathit{exp} \\
 & \mid & \texttt{store}\ \mathit{type}\ \mathit{exp}\texttt{,}\ \mathit{type}\ \texttt{*}\ \mathit{exp} \\
 & \mid & \texttt{br i1}\ \mathit{exp}\ \texttt{, label}\ \mathit{lbl}\ \texttt{, label}\ \mathit{lbl} \\
 & \mid & \texttt{br label}\ \mathit{lbl} \\
 & \mid & \texttt{ret}\ \langle \mathit{type}\ \mathit{exp} \mid \texttt{void}\ \rangle \\
\mathit{exp} & ::= & \mathit{name} \mid \mathit{iliteral} \mid \texttt{getelementptr (}\ \mathit{type}\ \mathit{exp}\ \texttt{,}\ \mathit{index}\ \texttt{,}\ \mathit{index}\ \texttt{)} \\
\mathit{index} & ::= & \mathit{itype}\ \mathit{iliteral} \\
\mathit{branch} & ::= & \mathit{iliteral}\ \mathit{iliteral}\ \mathit{lbl} \\
\mathit{arg} & ::= & \mathit{type}\ \mathit{exp}
\end{array}
$$

Fig. 2. Grammar of the target LLVM IR subset: *itype*, *iliteral*, *lbl* and *name* correspond respectively to integer types, integer literals, labels and names. Choices are separated by | and optionally delimited by ⟨ and ⟩. The $^{+}$ superscript denotes a comma-separated list of elements of the annotated entity.

`i8 }` is the type for structures with three fields, the first having as type pointer to `i8`. Grammar rule *type_def* states how types are named, e.g., `%T1 = type {i32, i32}` and `%T2 = type {%T1*, %T1*}`. In LLVM IR, pointer values are integers.

Local entities are constants, variables or functions. An example of constant definition is `@secret = constant i32 42` and is composed of a name, type and value. A variable definition has a name, a type and code generation attributes, e.g `@count = common global i32 zeroinitializer`. Attributes provide information for target code generation, e.g., linkage type, scope, initialization. For each such definition, a memory block is allocated statically and stores the variable value. Function definitions are composed of the signature and body. The signature contains the return type, name, parameters, and attributes for target code generation. The body is a sequence of blocks of instructions in single-static assignment form.

Grammar rule *inst* describes the different kinds of instructions. All instructions producing a value assign it to a fresh variable (since it is a SSA language).

Instruction `alloca` allocates a memory block, with the size of the given type, on the stack segment. This memory is automatically freed when the current frame is popped from the stack. Arithmetic operations are binary and comparisons return a 1-bit integer value. Instruction `call` invokes the given function with the given arguments, assigning the result to a fresh variable. In general, `getelementptr` gets the address of an element in an aggregate object through indexing. This instruction assumes that a sequence with several aggregate values may be stored starting from the given position. It therefore gets two indices: the first identifies which value is selected in the sequence, and the second selects the element of interest in the aggregate. In the LLVM IR code produced by b2llvm, such sequences are composed of a single structure value. Hence, the first index has value `0` (and type `i32`) to select the first structure at the given location *exp*, and the second index selects a field in that structure. Instruction `load` assigns to a fresh variable *name* the contents of a memory address of type *type* specified by *exp* (e.g. in figure 1). Instruction `store` writes a value to memory address (e.g., see figure 1). Instruction `br` is either conditional, and directs the execution to one of two blocks, or unconditional and the execution jumps to the given block. Instruction `switch` directs the control flow to one of several blocks, according to the value of the given expression. Finally, instruction `ret` ends the current function call, optionally returning a value. The expression language is thus limited to names (local and global), integer literals and selection of an element in a structure.

We make no assumption on the existence of a library to obtain resources managed by the operating system, such as dynamic memory allocation. Consequently, all data must be allocated either statically, or on the current stack frame (using the `alloca` instruction).

3 On the structure of B developments

Industrial applications of the B-method are large-scale developments that use constructs for modular design. The b2llvm code generator supports these constructs. We discuss them in this section.

A B *project* consists in specifying a system at an abstract level and in deriving a consistent software system. This is essentially done by decomposing the specification into *modules* and by producing computer-executable artifacts from such modules. In a B development, software is organized in libraries of *modules* which may be composed to build new modules and realize projects.

A module has a specification, called a *machine*, and is developed formally by a series of modules called *refinements*. Such modules may be used to specify additional requirements, to define how abstract data may be encoded using concrete data types, or to define how operations may be implemented algorithmically. From a formal point of view, each module is simulated by the subsequent refinement modules. A refinement is called an *implementation* when its data is scalar and behavior is described in a procedural style. Implementations may be translated into an imperative programming language such as C.

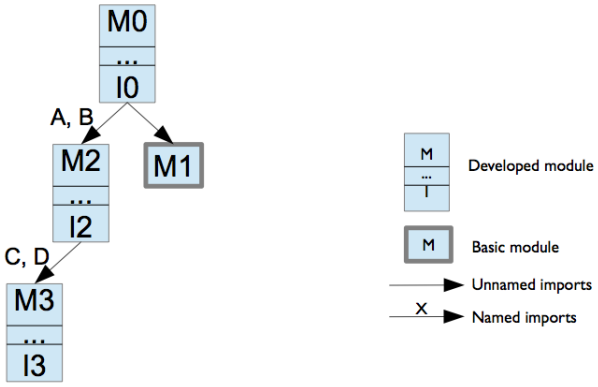

Fig. 3. Structure of the imports relation in a B project

Machines, refinements and implementations are called the *components* of a module. When a module is implemented with the B-method, it is called a *developed* module. It is also possible that a module is only specified, but not implemented, in B. It is then called a *base* module. b2llvm handles projects with both kinds of modules.

Among the modularity constructs in the B notation, handling the *import* relation requires the application of separate compilation techniques. At the implementation level, one module may import several instances of a module (base or developed) to form its internal data structures. The implementation of an imported developed module may in turn import other instances, and there is no pre-established limit to such chain of imports. The import relation between module instances forms a tree, where the root is an implementation and the descendants of a node are the instances imported from the module in that node. Figure 3 shows the structure of a B project with an implementation I0 of a specification M0. I0 imports one unnamed instance of base module M1 and two instances named A and B of developed module M2. Its implementation I2 in turn imports two instances C and D of a developed module M3, implemented as I3 (which has no imports itself).

4 General design of the code generator

The input to b2llvm is a large subset of the B implementation language, also known as B0: simple data types INT and BOOL, enumerations, concrete variables, concrete constants, *sees* clause, importation (i.e., instantiation) of modules, and all instructions, including operation calls. Support for arrays and record types is also underway and will be integrated to the code generator. This input is given as XML-formatted files produced by Atelier-B version 4.2. In addition to producing LLVM IR from B implementations, b2llvm is designed to satisfy the following two requirements:

static memory allocation Many safety-critical systems preclude the use of dynamic memory allocation. Therefore, all memory has to be allocated statically, save for the function call frame stack. As a side effect, no dynamic memory allocation library is required.

separate compilation An internal change in a module should only require generating new IR code for that module, and not for the modules that depend on it. This condition is important for large projects, where the development may be distributed. Nevertheless, a change in an interface still requires the recompilation of all dependent modules.

We will use the example of the B project structure from figure 3 to explain decisions taken in the design of the code generator. This project consists of four modules and the corresponding final binary must include one instance of module M0, one instance of M1, two instances of M2 and four instances of M3.

M1 is a base module, and the corresponding instance is produced by another tool chain. Nevertheless, M0 may use all the symbols defined in the interface of M1 and we have to include corresponding declarations in the LLVM IR file for M0. Similarly, M0 accesses the interface of M2, and M2 accesses that of M3. The first design decision is that, given a module M, we need a procedure producing the LLVM IR declarations for all the elements in the interface of M. The code thus produced is called the *interface section* of (the translation of) M.

Next, modules may have a data space and the code generator needs to allocate memory to store the representation of the corresponding data. Dynamic memory allocation is excluded, and the sole solution is static memory allocation, that is using global variables. Also, when the code generator processes a B module, the number of its instances at run time is unknown, and we would not want to have to regenerate code each time the module is used in a project to suit the number of instances. So, the second design decision is to distinguish between code generation of a module and code generation of its instances. To cope with it, the b2llvm code generator has two operation modes:

- *COMP*, for module compilation, consists in producing an LLVM IR implementation of the data, i.e., a type encoding the state space, and of the behavior, i.e., functions implementing initialization and operations.
- *PROJ*, for module instances, is applied whenever we need to instantiate modules, that is, when we want to produce code for a full project. Then, given the root module of the project, all transitively imported components are identified and instantiated, by generating LLVM IR global variables having the type associated with the corresponding module. Note that the definitions of such module types are generated in COMP mode.

The code generated for a module with a data space needs to address individually the variables and the imported module instances composing such a space. To do so, these are aggregated within a structure-like data type. Hence, when a module has a data space, b2llvm produces the definition of a LLVM IR structure type, named `%M$state$`. This definition is called the *typedef section* of (the translation of) M.

To support separate compilation, the representation of the imported module instances cannot be part of the structure itself. Instead, the instances of the imported modules are represented as references to the corresponding encoding structures (i.e., as pointers). We call `%M$ref$` the type pointer to `%M$state$`.

typedef: If the module has data space, a LLVM IR structure type is defined:

```
%M$state$ = type { type+ }
```

interface: If the module has a data space, an LLVM IR type M_ref, pointer to M_data and an initialization function are defined:

```
%M$ref$ = type %M$state$*
declare void @M$init$(%M$ref$, type+)
```

One function is declared for each operation in the module:

```
declare void @M$op(%M$ref$, type+)
```

implementation: For developed modules, defines the functions declared in the interface:

```
define void @M$op(%M$ref$ %self$, param+) {
    block+
    exit: ret void
}
```

Fig. 4. Summary of the different sections and the pattern of LLVM IR code composing them.

To encode the behavior of a module, for each operation op, a function named `@M$op` encoding its behavior is generated. The parameter list of such functions contains one item for each input and output of the corresponding operation. There is also one parameter of type `%M$ref$`, which is a reference to the structure encoding the instance associated with that operation. Also, b2llvm produces a function `@M$init$` responsible for executing the initialization of M. The parameters of this function are the addresses of the instances found in the import tree of the module (including the module itself). These parameters are necessary to call the corresponding initialization functions in the correct dependency order and to bind the references to the imported modules to elements of the structure of the initialized module. These LLVM IR function definitions and the definition of the type `@M$ref$` form the so-called *implementation section* of (the translation of) M. Figure 4 summarizes the three sections defined in our approach for the code generation and figures 5 and 6 present the overall structure for the code generated in *COMP* mode and *PROJ*, respectively.

5 Details of the code generator

We have specified the code generation process with a comprehensive set of formal rules. Due to space constraints, we cannot thoroughly present this specification[2]. Instead, we give an informal description of the code generation process, based

[2] This specification is available online at http://www.b2llvm.org/b2llvm/downloads.

for each transitively imported stateful module Q, *generate*
    `%Q$state$ = type opaque` *(declares type for* Q *state space)*
    `%Q$ref$ = type %Q$state$*` *(and corresponding pointer type)*
for each imported module Q, *generate*
    include the interface section of Q
if M *is stateful*
    include the typedef section of M
    `%M$ref$ = type %M$state$*`
include the implementation section of M

Fig. 5. Code template for the *COMP* mode.

for each transitively imported stateful module Q
    include the typedef section of Q
    `%Q$ref$ = type %Q$state$*`
for each stateful instance Q, *imported transitively through* path
    declare a variable of type `%Q$state$`:
    `@Q[path] = common global %M$state$ zeroinitializer`
include the interface section of M
define a function `%$init$` *with a call to the*
    initialization function of M *with the proper bindings*
    `define void @$init$(void) {`
        `call void @M$init$(@M, { instances+ }) {`
        `exit: ret void`
    `}`

Fig. 6. Code template for the *COMP* mode.

on two examples: first, a counter with no external dependencies and, second, part of a watchdog timer that includes one instance of the same counter. We complete this section by describing an example of code generation for a project.

5.1 The standalone module counter

The implementation counter_i of the module counter is presented in figure 7.

```
1  IMPLEMENTATION counter_i
2  REFINES counter
3  CONCRETE_VARIABLES value, error
4  INVARIANT value: INT & error : BOOL & /* omitted gluing invariant */
5  INITIALISATION  value := 0; error := FALSE
6  OPERATIONS
7      inc = IF value < MAXINT THEN value := value + 1
8            ELSE error := TRUE END;
9      res <-- get = res := value
10 END
```

Fig. 7. Example B implementation.

Here, the code generator needs to access neither the corresponding B machine, nor the gluing invariant of the implementation. This module is stateful as

it has two state variables value and error (respectively an integer and a Boolean) and two operations inc and get. Figure 8 contains its corresponding typedef section: LLVM IR aggregate type `%counter$state$` has two elements, a `i32` at position 0 represents value and a `i1` at position 1 represents error.

```
1  %counter$state$ = type {i32, i1}
```

Fig. 8. Corresponding LLVM IR typedef section.

Figure 9 contains the corresponding interface section, comprised of the declarations of all the entities defined in the module that may be used by third-party components (this is illustrated in section 5.2): a pointer type `%counter$ref$` to reference an aggregate storing the state of the component, the initialization function `%counter$init$`, and the functions `%counter$inc` and `%counter$get`, each responsible for implementing one module operation. Each such function takes as first parameter the address of the representation of the module state. The last function also takes as parameter the address of a `i32`, where the value of the operation value is stored.

```
1  %counter$ref$ = type %counter$state$*
2  declare void @counter$init$(%counter$ref$)
3  declare void @counter$inc(%counter$ref$)
4  declare void @counter$get(%counter$ref$, i32*)
```

Fig. 9. Corresponding LLVM IR interface section.

Figure 10 contains the implementation section. It consists of the definition of all the functions implementing the module behavior. All function bodies contain an `entry` and an `exit` statement block. In addition, a block for each conditional branch is created; e.g., blocks starting line 18 and 25 respectively correspond to the IF branches from line 7 and 8 in the inc operation.

This example illustrates the encoding for different kinds of expressions and instructions. First, let us consider expressions: the example given in figure 7 includes operation parameters, integer and Boolean literals, implementation (state) variables, an addition and a comparison.

Operation parameters are encoded as function arguments, which have identifiers in LLVM IR. So the b2llvm code generator simply maintains a symbol table mapping each B operation parameter to the identifier of the corresponding LLVM IR function parameter. For instance, operation get has output res (l. 9, fig. 7), which is represented by `%res` of function `@counter$get` (l. 31, fig. 10). Integer and Boolean literals are encoded directly as 32-bit and 1-bit LLVM integer values; for instance MAXINT and TRUE are encoded respectively as `2147483647` and `1` (l. 15 and l. 26, fig. 10). Implementation variables are encoded as elements of the aggregate `%self$`, which is a parameter in each function. Their address is obtained with the `getelementptr` instruction, giving the position of the variable representation in this aggregate: `0` for variable value and `1` for variable error.

We now provide detailed explanation for the translation of the assignment value := value + 1 (l. 7, fig. 7) to LLVM IR instructions (l. 18-22, fig. 10):

```
1  define void @counter$init$(%counter$ref$ %self$) {
2  entry:
3    %0 = getelementptr %counter$ref$ %self$, i32 0, i32 0
4    store i32 0, i32* %0
5    %1 = getelementptr %counter$ref$ %self$, i32 0, i32 1
6    store i1 0, i1* %1
7    br label %exit
8  exit:
9    ret void
10 }
11 define void @counter$inc(%counter$ref$ %self$) {
12 entry:
13   %0 = getelementptr %counter$ref$ %self$, i32 0, i32 0
14   %1 = load i32* %0
15   %2 = icmp slt i32 %1, 2147483647
16   br i1 %2, label %label0, label %label1
17 label0:
18   %3 = getelementptr %counter$ref$ %self$, i32 0, i32 0
19   %4 = load i32* %3
20   %5 = add i32 %4, 1
21   %6 = getelementptr %counter$ref$ %self$, i32 0, i32 0
22   store i32 %5, i32* %6
23   br label %exit
24 label1:
25   %7 = getelementptr %counter$ref$ %self$, i32 0, i32 1
26   store i1 1, i1* %7
27   br label %exit
28 exit:
29   ret void
30 }
31 define void @counter$get(%counter$ref$ %self$, i32* %res) {
32 entry:
33   %0 = getelementptr %counter$ref$ %self$, i32 0, i32 0
34   %1 = load i32* %0
35   store i32 %1, i32* %res
36   br label %exit
37 exit:
38   ret void
39 }
```

Fig. 10. LLVM implementation section for counter_i.

l. 18-20 First, the right-hand side of the assignment is evaluated, and the result is stored in temporary %5, as follows:

- **l. 18** A `getelementptr` instruction gets the address of the representation of variable value in the structure encoding the state of the module, and the result is stored into temporary %3.
- **l. 19** A `load` instruction fetches the data in this location into temporary %4.
- **l. 20** An add instruction sums this value with one (1) and the result is stored into temporary %5.

l. 21 Second, the left-hand side of the assignment is evaluated, the result being stored in temporary %6. Since this is again the variable value, it is essentially the same operation as in l. 18 and %6 is redundant with %3 (but we do not deal with optimization at this stage).

l. 22 Finally, the assignment effectively take place with the value in %5 begin stored at the evaluated address %6 (that of the representation of variable value).

For an IF instruction (e.g., lines 7-8, fig. 7), b2llvm generates code to evaluate the condition (e.g., lines 13-15, fig. 10), a conditional branch (l. 16, fig. 10), and

one block with the encoding of each branch (l. 17-23 and 24-27, fig. 10). Note that each such block must end with an unconditional branch to the instruction following the conditional. b2llvm handles the creation of all the required block labels.

Of course, the code thus generated is not optimal: e.g., the exit block in the initialisation is useless. Indeed, code generation is designed to be as simple as possible. No optimizations are implemented into b2llvm. A positive consequence of choosing LLVM as target architecture is the possibility of applying many off-the-shelf optimizers developped for LLVM to the output of b2llvm.

5.2 The composed module wd

Our second example is the wd module detailed in figure 11. It contains module

```
IMPLEMENTATION wd_i
REFINES wd
VALUES timeout=50
IMPORTS counter
INVARIANT overflow = FALSE & timeout - value = ticker
INITIALISATION
   VAR count IN
       count := 0;
       WHILE count < timeout DO
           inc; count := count+1
       INVARIANT value = count
       VARIANT timeout - count
       END
   END
OPERATIONS
   tick =
   VAR elapsed, diff IN
       elapsed <-- get;
       diff := timeout - elapsed;
       IF diff > 0 THEN inc END
END;
```

Fig. 11. Implementation of B module wd (excerpts).

instantiation, operation calls, and a loop instruction. The state of this module is exactly the state of the unique instance of its counter component, and is encoded as an aggregate with a unique element, of type pointer to the state representation of the corresponding module (see typedef section in figure 12).

```
1  %wd$state$ = type {%counter$ref$}
```

Fig. 12. Typedef section for module WD.

Functions `@wd$init$` and `@wd$tick` are defined in the implementation section, presented in figure 13. We first discuss the latter.

Function `@wd$tick$` implements operation tick. It has no input and no output. Its sole argument is the address of the representation of a wd instance. It has two local variables, elapsed and diff, and both are integers. Operation variables are represented in stack memory, which is reserved with the `alloca` instruction (e.g., lines 28-29). LLVM requires that such allocations appear first

```
define void @wd$init$(%wd$ref$ %self$, %counter$ref$ %arg0$) {
entry:
  %count = alloca i32
  %0 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  store %counter$ref$ %arg0$, %counter$ref$* %0
  call void @counter$init$(%counter$ref$ %arg0$)
  store i32 0, i32* %count
  br label %label1
label1:
  %1 = load i32* %count
  %2 = icmp slt i32 %1, 50
  br i1 %2, label %label2, label %label0
label2:
  %3 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  %4 = load %counter$ref$* %3
  call void @counter$inc(%counter$ref$ %4)
  %5 = load i32* %count
  %6 = add i32 %5, 1
  store i32 %6, i32* %count
  br label %label1
label0:
  br label %exit
exit:
  ret void
}
define void @wd$tick(%wd$ref$ %self$) {
entry:
  %elapsed = alloca i32
  %diff = alloca i32
  %0 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  %1 = load %counter$ref$* %0
  call void @counter$get(%counter$ref$ %1, i32* %elapsed)
  %2 = load i32* %elapsed
  %3 = sub i32 50, %2
  store i32 %3, i32* %diff
  %4 = load i32* %diff
  %5 = icmp sgt i32 %4, 0
  br i1 %5, label %label1, label %label0
label1:
  %6 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  %7 = load %counter$ref$* %6
  call void @counter$inc(%counter$ref$ %7)
  br label %label0
label0:
  br label %exit
exit:
  ret void
}
```

Fig. 13. Implementation section for module WD.

in function bodies. The procedure in b2llvm responsible for encoding B operation declarations has a dedicated preliminary pass that collects all local variables and issues the corresponding allocations. The encoding of operation calls is illustrated lines 30-32. First, the address of the called operation module is computed (lines 30-31), possibly followed by the computation of other parameters. Then a LLVM function call is issued (e.g., l .32). Notice that the output of the operation is stored on the stack at the location given by parameter `%elapsed`. Another example of operation call is given in lines 40-42. The remaining code in the function body uses previously described techniques.

Function `@wd$init$` implements the initialization of a `wd` instance, the address of which is given in parameter `%self$`. It also gets the address of one module instance for each component. In this example, there is one instantiation of module counter and its representation is given through parameter `%arg0$`. Following an allocation for the representation of variable count, the components are bound and initialized: each component representation is bound to an element of the aggregate representing the current instance (e.g., lines 4-5). Also, each component representation is initialized by calling the corresponding LLVM function (e.g., line 6). The order of these initializations complies to the dependency order. This example illustrates code generation for loops. First the loop condition is evaluated (lines 10-11 in the example), and a conditional branch jumps either to the code implementing the loop body, or to the first instruction after the loop encoding (e.g., line 12). Also, the loop body ends with an unconditional branch to the block evaluating the loop condition (e.g., line 20).

5.3 Generating a system instance

To conclude this section, we demonstrate and discuss the result of the code generation in PROJ mode. For this, we take the wd module as example and consider the resulting code in figure 14. It follows the template given in figure 6 and is composed of the following: lines 1-2 are the type definitions of the imported component and line 3 is the global variable corresponding to its sole instance, then lines 4-8 correspond to the interface section of the root component (comprised of the declarations of types and functions), and concluded by the definition a system initialization function called `@$init$` (lines 9-13). The role of this function is to call the initialization function of the top-level module passing it the state-representation variables as parameters.

6 Verification and validation

Considering the verification of the proposed translation, several approaches are possible: inspection, testing and proof. Only the first is currently available, and work is underway to provide a framework based on the second approach.

A first approach is based on human inspection of the generated code. Currently, b2llvm provides two options to assist such inspections. The first option

```
%counter$state$ = type {i32, i1}
%counter$ref$ = type %counter$state$*
%wd$state$ = type {%counter$ref$}
%wd$ref$ = type %wd$state$*
@$wd = common global %wd$state$ zeroinitializer
@$counter = common global %counter$state$ zeroinitializer
declare void @wd$init$(%wd$ref$, %counter$ref$)
declare void @wd$tick(%wd$ref$)
define void @$init$() {
entry:
  call void @wd$init$(%wd$ref$ @$wd, %counter$ref$ @$counter)
  ret void
}
```

Fig. 14. Code generation in *PROJ* mode.

provides additional functions to output the values of state variables of the system: they can be called to visualize the evolution between operation calls. The first option consists in annotating the code generated with information on the intent of the code and references to the original B implementation. Figure 15 contains an excerpt of such annotated LLVM IR.

A second approach is runtime verification, by application of simulation, test and assertions. For instance, we can use existing approaches to generate tests [2,11] from the B development artifacts. Such tests can then be converted to LLVM IR to produce and run test harnesses. A typical scenario for such a test would consist in setting the state variables to some given values, call the function implementing the operation being verified, and then inspect these same variables to check if they have the expected values. Work in this direction, based on [11] is currently underway.

It would also be possible to perform dual animation: B model with ProB [10] together with the generated code with the LLVM interpreter. In addition, B artifacts contain several kinds of assertions: state invariants, loop invariants, preconditions, as well as *ad hoc* assertions. If the conditions found in those assertions are expressed in the B0 language, b2llvm can also translate them to LLVM IR. Using them to monitor the generated code at run-time would additionally require to provide an LLVM implementation of the assertion semantics (e.g., the assert command found in the standard C library) and link the generated code to this implementation.

Finally, to formally verify the correctness of the translation, the semantics of B and LLVM IR need to be specified in a suitable framework. Vellvm [12] is a Coq formalization of the LLVM IR semantics that uses CompCert's memory model [9]. This could be a starting point for a proof of the correctness of the translation rules. It would still require to formalize the semantics of B and of the translation rules in that same setting. This is left as future work.

7 Conclusion

This paper presents an approach to generate executable code from a large subset of the B implementation language using LLVM, a modern compiler design infrastructure. This is a work in progress, yet the definition is self-contained and

```
;;1 The type for the state of "counter" is defined in "counter_i",
;; it is an aggregate such that:
;;1.1 Position "0" represents variable "value".
;;1.2 Position "1" represents variable "error".
%counter$state$ = type {i32, i1}
;;2 The type for references to state encodings of "counter" is:
%counter$ref$ = type %counter$state$*
;;3 The function implementing initialisation for "counter" is
;;  named "@counter£init£" and has the following parameters:
;;3.1 "%self£": address of LLVM aggregate storing state of "counter";
define void @counter$init$(%counter$ref$ %self$) {
```

Fig. 15. Excerpt of annotated LLVM code generated by b2llvm.

has a large enough scope to be applied to B implementations where the data belongs to basic types. Its implementation, called b2llvm, is already publicly available and will eventually be distributed as an extension to Atelier-B under an open-source license.

Our current work is to extend the scope to the full B implementation language. This entails the inclusion into the translation of rules to handle aggregate data types as well as some syntactic sugar. We are also planning for producing a LLVM IR output with debugging information. Such output would be indeed very helpful to provide feedback to the user when applying testing to validate the produced code.

To prove the correctness of the translation, we would have to define the semantics of B and LLVM IR in a unified framework. Possible starting points are Vellvm [12], a framework to reason about the correctness of LLVM programs and transformations, and the existing formalizations of the B method (e.g., [7,3,4]). We would have to extend such a framework to encompass both B and LLVM IR. Another possible approach would be to translate verification conditions from the B development artifacts as assertions in the generated LLVM IR. The compiled program would include checks that such assertions hold while executing.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.
2. F. Ambert, F. Bouquet, B. Legeard, F. Peureux, and al. BZ-Testing-Tools, 2002.
3. J.-P. Bodeveix, M. Filali, and C. Muñoz. A formalization of the B-method in Coq and PVS. In *Electronic Proc. B-User Group Meeting FM 99*, pages 33–49, 1999.
4. P. Chartier. Formalisation of B in Isabelle/HOL. In *Proc. B'98*. Springer, 1998.
5. ClearSy. ComenC, B0 implementation translation into C language. Available at:http://www.comenc.eu, 2008.
6. ClearSy. *Atelier B User Manual Version 4.0*. Clearsy System Engineering, 2009.
7. É. Jaeger and C. Dubois. Why would you trust B ? In *LPAR 2007*, volume 4790 of *LNCS*, pages 288–302, 2007.
8. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

9. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
10. M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.
11. E. Matos and A. Moreira. A B based testing approach. In *Formal Methods: Foundations and Applications*, number 7498 in LNCS, pages 51–66. Springer, 2012.
12. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.