ALF (ARTIST2 Language for Flow Analysis) Specification

Andreas Ermedahl, Jan Gustafsson, and Björn Lisper
School of Innovation, Design, and Engineering, Mälardalen University, Västerås, Sweden
{andreas.ermedahl,jan.gustafsson,bjorn.lisper}@mdh.se

VERSION: 1.31

Abstract

ALF (ARTIST2 Language for Flow Analysis) is a language intended to be used for flow analysis in conjunction with WCET (Worst Case Execution Time) analysis. ALF is designed to be possible to generate from a rich set of sources: linked binaries, source code, compiler intermediate formats, and possibly more.

Changes since Latest Version (1.30)

This list does not include minor clarifications and similar, only changes that affect the syntax and/or the semantics of ALF.

- The signatures of the shift operations (`l_shift`, `r_shift`, `r_shift_a`) have been changed to demand separate sizes of the two operands (bitstring to shift, and no. of positions to shift).
- Labels that constitute function names must have offset zero. This was not clearly stated in the previous version of the ALF specification.
- The semantics for concurrent assignment has been made more precise: in case of write conflicts (several address expressions evaluate to the same address), it is nondeterministically chosen which value that gets written.
- The semantics for switch statements has been changed to disallow implicit fall-through. This makes possible a more precise modelling of program flows in cases where the ALF generator has the knowledge that a set of program branches is complete. An ALF analysis tool can then assume that the fall-through cannot be taken, and exclude that path from the analysis. The change in semantics also makes it possible to encode asserts in ALF using switch.
- *CHAR_STRING* can now be any C string literal. C string syntax is widely accepted as a standard, and we saw no reason to deviate from it in ALF.
- Identifiers can now also be C string literals. This is to allow easy encoding/decoding of "weird" identifiers, used by linkers and similar, when translating to/from ALF.
- These updates concern the syntax for macros, which are not yet in use in any ALF tool:
  - In the production rule for *ACTUAL* , *LABEL_STMT*+ has been replaced by *STMTS* and *STMT* by *LABEL_STMT* .
  - Similarly, In the production rule for *DEFINABLE* , *LABEL_STMT*+ has been replaced by *STMTS* and *LABEL_STMT* .
  - Macro rules have been introduced for *FLOAT_VAL* in the ALF grammar. The previous omission was a simple oversight.

1 Background and Introduction

ALF is an intermediate format which is designed to be amenable to program analysis rather than code generation. In that respect, it is therefore different from most compiler intermediate formats. ALF is designed to be possible to generate from a rich set of sources: linked binaries, source code, compiler intermediate formats, and possibly more. This has certain implications for ALF's program model, which must encompass both high- and low-level constructs while being as amenable to program analysis as possible. ALF will be used as input format for the Worst-Case Execution Time (WCET) analysis tool SWEET (SWEdish Execution Time tool).

ALF is basically a sequential imperative language, with a memory model that differs between program and data. Only data can be modified: thus, self-modifying programs cannot be modeled in ALF in a direct way. The memory model and instruction set supports dynamic jumps as well as both static and dynamic allocation of data areas.

ALF has a fully textual representation: it is not, as for some intermediate formats, based on an accompanying CFG representation, or represented as a data structure in some programming language. It can thus be seen as an ordinary programming language, although it is intended to mainly be generated by tools rather than written by hand.

1.1 The Structure of an ALF Program

An ALF program consists of the following declarations, in the following order:

Least-adressable-unit-declaration – specification of size, in bits, of the Least Addressable Unit (LAU). (For both data and code memory, the underlying assumption being that they are both equal)

Endianness-declaration – specification of little/big-endianness

Export-declarations – declaration(s) of exported symbols

Import-declarations – declaration(s) of imported symbols

Allocations – allocation of static data areas

Initializations – possible initialization of static data areas

Volatile-declarations – declaration(s) of memory addresses for volatile data (which can change outside the control of the program)

Function-declarations – function (procedure) declaration(s), possibly including a "main" procedure which then will provide the global entry point to the program

1.2 Syntax

ALF has a Lisp/Erlang-like syntax, to make it easy to parse. This syntax uses prefix notation as in Lisp, but with curly brackets "{", "}" as parentheses as in Erlang. An example is

```
{ dec_unsigned 32 2 }
```

which denotes the unsigned 32-bit constant 2. The syntax will be used when exemplifying ALF concepts and constructs below.

ALF can easily be given an XML syntax, if desired. For the moment, we prefer the Lisp/Erlang style since we think it's more readable and also easy enough to parse.

1.3 Memory Model

ALF's memory model distinguishes between program and data addresses. It is essentially a memory model for relocatable, unlinked code. Program and data addresses thus both have a symbolic base address, and a numerical offset. Two addresses are equal only if they have the same base address, and the same offset[1]. The address spaces for code and data are disjoint.

[1] Addresses also have a *size*, specifying how much memory they occupy when stored. In order to be equal, two addresses must also have the same size, see Sections 1.4 and 1.10

1.3.1 Program Model

ALF's program model is quite high-level, and similar to C. An ALF program is divided into a number of functions, corresponding to function declarations. Within each function, the program is a linear sequence of statements, with execution normally flowing from one statement to the next. The last executed statement must be either a return statement, or the last statement in the function body. Some of the statements may be tagged with symbolic labels, which consist of a symbolic base pointer (label reference, or lref), and a numerical offset. Jump or switch statements (see Section 1.6) can direct the control to a labelled statement in the same function.

Each function is labelled, and this label can be used to call the function from another function, or recursively from the function itself. Function labels are distinct from code labels: it is an error to call a code label, or to jump to a function label.

The target label in a call, jump, or switch can be dynamically calculated, either through computing the numerical offset of the label, or by loading the label from data memory.

Functions take arguments, return one or several values, and can have local variables: functions cannot, however, be declared inside other functions. ALF is lexically scoped, and locally defined variables, or formal arguments, take precedence over globally defined entities with the same name. ALF has no explicit stack, but can be given an operational semantics which uses a stack for arguments, results, and local variables. ALF uses call-by-value left-to-right evaluation order of function arguments.

The run-time representation of ALF statements is not specified, and there is no way an ALF program can access or alter its own code. If an ALF program models an actual program for a Harvard processor and the actual code memory contains, e.g., constant data that the actual program reads, the ALF program must model that constant data as ALF constants or as residing in a part of the ALF data memory.

1.3.2 Data Model

ALF's data memory is divided into *frames*. Each frame has a symbolic base pointer (frameref) and a size specified in bits. Frames can also be given an unbounded size, which can be useful when modeling dynamic data areas such as heaps, or stacks. Like labels, data addresses are formed from a symbolic part and an offset. The symbolic part of a data address is a *frameref* (see below), and the offset is a natural number in the *least adressable unit* (LAU) of the ALF program, which typically is a byte (8 bits) but also can be chosen differently.

The requirement that addresses with different base addresses (framerefs) are distinct implies that frames in ALF are non-overlapping. This can be used to improve the precision when analyzing ALF code. On the other hand, if the ALF frames represent data areas that indeed can be overlapping then the analysis is potentially unsound. It is the responsibility of the ALF translator to ensure that this won't happen, alternatively that the user is aware of the potential problem. Future versions of ALF may become equipped with some means to specify that certain frames may be overlapping.

Frames can be either statically or dynamically allocated. Statically allocated memory is explicitly declared. There are two ways to allocate memory dynamically:

- As local data in so-called *scopes*. An example of a scope is a function body. This kind of local data is declared similarly to statically allocated data, but in the context of a scope rather than globally for the main program. It could be allocated on a stack in an ALF implementation.
- By calling a special function that returns a reference to a newly allocated frame. This kind of data would typically be allocated on a heap.

In order to model and analyze accesses to especially the second kind of dynamically allocated frames, we assume that a frameref is a pair (a, i), where a is an identifier and i is a natural number. One way to think of it is that, for each identifier a, we have an infinite array of framerefs $a[i]$. This can be used to model programs with several areas of dynamically allocated memory: one array of framerefs is then associated with each area.

For each identifier used in a frameref, a counter is associated. When a frame is allocated an identifier id is referenced: a frameref (id, n) is then returned, where n is the current value of the counter, and the counter is then incremented. This ensures that dynamically allocated frames always have distinct framerefs. (Cf. `malloc` in C.)

Another reason to index framerefs with natural numbers is that they then can be analyzed statically with, say, an interval analysis. Thus, it will sometimes be possible to, e.g., find that different accesses always will go to different frames. This can improve the precision of subsequent program analyses.

Parts of statically declared frames can optionally be initialized. The initialization may also declare such parts as *volatile*, or *read-only*. A part of a frame being volatile means that its contents may change at any time, in a way not under the control of the program. A program analysis will have to take this into account. Conversely, a read-only declaration can be used by a program analysis as an assertion that the contents of the declared memory area will not change. Usually a write to an address declared read-only should be considered an error: however, for WCET analysis it might be appropriate to also represent the initalization code in ALF, and this code will of course then write to the initialized addresses. For apparent reasons the same memory area cannot be declared volatile and read-only at the same time.

A data memory model like this, with data addresses $((id, i), \mathit{offset})$, can be used both for high-level code, intermediate formats, and reverse-engineered binaries. Here are some examples how to do it:

- "High-level": allocate one frame per high-level data structure in the program (struct, array, ...)
- "Intermediate-level": use one identifier for each stack, heap etc. in the runtime system, in order to model it by potentially unbounded "array" of frames (one for each object stored in the data structure). Use one frame for each possible register. If the intermediate model has an infinite register bank, then use one identifier to model the bank (again one frame per register).
- Reverse-engineered binaries: allocate one frame per register, and a single frame for the whole main memory. (For a given processor architecture, this can be done statically, by some standard preamble included in the beginning of the ALF code.)

A language like C poses a particular problem in that it allows to use absolute addresses for pointers while the addresses to data areas for program variables are not known until after linking. Since ALF's memory model does not include absolute addresses, a C-2-ALF translator will have to allocate a special frame to host the absolute addresses. This causes a potential problem with aliasing since the linker may put user data, represented by other ALF frames, onto the same addresses. As mentioned, ALF's memory model considers different frames to be non-overlapping so this can lead to an unsound analysis. However, most often the absolute adresses will be used to access i/o ports and similar where no user data will be allocated, and then the analysis will be sound.

1.4 Values

Values can be:

- numerical values: signed/unsigned integers, floats, etc.,
- framerefs,
- lrefs,
- data addresses (f, o), where f is a frameref and o is an offset (natural number), or
- code adresses (labels) (f, n), where f is an lref and n a natural number.

There is a special value `undefined`. This provides a fallback in situations where an ALF-producing tool, e.g., a binary-to-ALF translator, cannot translate a piece of the binary code into sensible ALF. Translators to ALF should use `undefined` only when necessary, since its appearance can reduce the precision of an analysis. If the ALF code is analyzed with a tool that allows value annotations then it can sometimes be more appropriate to use such annotations instead, since this avoids "hardwiring" the undefined semantics for the ALF code.

Each value has a type (see Section 1.10). Each type has a maximal *size* (in bits): this is the number of bits used to store the value in memory. The size of a value is mostly given explicitly. ALF has unbounded integer types whose size is infinity: values of these types are used to give semantics for operations which are not predefined in ALF. Values of bounded type are *storable*: they can be loaded from and stored to ALF's data memory.

Values can only be equal if they have the same type (however, see Section 1.10). One implication of this is that values of different sizes can never be equal, since ALF types include the size. For instance, the constant "17" of size 16 bits is different from the constant "17" of size 32 bits.

Some languages assign a particular meaning to certain values. For instance, C has a particular null pointer that is invalid to dereference. ALF has no special support for such values: the standard ALF features must be used to represent them. To represent the null pointer, we recommend a distinct frame with a single offset 0.

1.5 Operators

Operators in ALF are of five kinds:

- *Operators on data of limited size*. These operators mostly model the functional semantics of common machine instructions (arithmetic and bitwise logical operations, shifts, comparisons, etc).
- *Operators on data of unbounded size*. These are "mathematical" operations, typically on integers: all "usual" arithmetic/relational operators have versions for unbounded integers, and ALF also has a two-exponent operator on unbounded natural numbers. They are intended to be used to model the functional semantics for instructions whose results cannot be expressed directly using the finite-size-data operators. An example is the settings of different flags after having carried out certain operations, where ALF sometimes does not provide a direct operator.
- *Operators on bitstrings*. They are intended to model the semantics on bitstring level, which is appropriate for different operations involving masking etc. There is a variety of such operations in different instruction sets, and it would be hard to provide direct operators for them all.
- A *conditional*. Such an operator is useful when defining the functional semantics of instructions.
- A *conversion function* from bitstrings to natural numbers. Also this function is useful when defining the functional semantics of instructions.

A representative example is

{ `add` W $VEXPR_1$ $VEXPR_2$ $CEXPR$ }

where W is an integer constant specifying the bitwidth of the arguments, and the result, $VEXPR_1$, $VEXPR_2$ are expressions for the arithmetic operands in, and $CEXPR$ is an expression specifying carry in.

The arithmetic/logic operators in general take one or several bitwidths as an argument. The rationale for this is that many such operations appear for many bitwidths, and that their semantics then often is easily parameterized w.r.t. the bitwidth as well. This is reflected in ALF's type system (see Section 1.10).

In addition to the stateless operators, ALF has a load operator which evaluates an address expression into an address (f, o), and returns the current value held at that address.

1.6 Statements

In addition to the allocation statements described in Section 1.3, ALF has the following statements, with their informal semantics given below:

{ `null` }

Do nothing.

{ `store` $ADDRESS\_EXPR+$ `with` $EXPR+$ }

Concurrent assignment: evaluate the address expressions in $ADDRESS\_EXPR+$ into $a_1, \ldots, a_n$, in left-to-right order, then evaluate the expressions in $EXPR+$ into $e_1, \ldots, e_n$ (same order), and concurrently store each e_i at address a_i. In case several target addresses a_i are equal, then it is nondeterministically chosen which e_i will be stored.

{ `switch` NUM_EXPR { `target` $INT\_NUM\_VAL_0$ $LABEL\_EXPR_0$ } ...
{ `target` $INT\_NUM\_VAL_{n-1}$ $LABEL\_EXPR_{n-1}$ } }

NUM_EXPR is evaluated, and then compared to each constant $INT\_NUM\_VAL_i$ in order. If the computed value is equal to the j:th constant $INT\_NUM\_VAL_j$, then execution continues at the label given by evaluating the label expression $LABEL\_EXPR_j$. A switch may also have `default` cases: the first one reached will unconditionally cause a jump to the result of evaluating its $LABEL\_EXPR$. Fall-through is not allowed: if no constant matches (and there is no `default` case), then execution fails. Implicit fall-through can be modeled with a final `default` case jumping to a label immediately after the switch statement.

A switch with one $LABEL\_EXPR$ (and a `default` case modeling implicit fall-through) is basically a conditional branch.

{ `jump` $LABEL\_EXPR$ `leaving` n }

Evaluate $LABEL\_EXPR$ and jump unconditionally to the resulting address. n is a nonnegative integer constant: it specifies how many *scope nesting levels* the jump may exit from the current *scope*. See Section 1.8.

{ `free` $FREF\_EXPR$ }

Evaluate $FREF\_EXPR$, and deallocate the dynamically (with `dyn_alloc`) allocated memory pointed to by the result.

{ `call` $LABEL\_EXPR$ $EXPR\_LIST$ `result` $ADDR\_EXPR\_LIST$ }

Evaluate $LABEL\_EXPR$ (the function to be called), the expressions in $EXPR\_LIST$ (the arguments), and $ADDR\_EXPR\_LIST$ (the addresses where to store the results), in that given left-to-right order, write each evaluated argument in $EXPR\_LIST$ to the corresponding formal argument for the procedure which $LABEL\_EXPR$ evaluated into, and then call this procedure (i.e., call-by-value). When the call returns, store the return value(s), in left-to-right order, into the address(es) resulting from the evaluation of $ADDR\_EXPR\_LIST$.

$EXPR\_LIST$ can be left empty if the procedure takes no arguments. $ADDR\_EXPR\_LIST$ can also be left empty, if the procedure returns no values. The number of arguments must be the same for the call statement and the called procedure, and the same goes for return values. Since the called procedure is dynamically determined, these properties must in general be checked at runtime. A violation results in a runtime error.

{ `return` $EXPR\_LIST$ }

Values and control are returned using a `return` statement, which takes a list of expressions to be evaluated, in left-to-right order, when the statement is reached by the execution. All return statements in a function must have equally long lists, and as noted above, the length of these must also equal the length of the $ADDR\_EXPR\_LIST$'s in the calls to the function.

A procedure named "main" will be executed when an ALF program runs. A runnable ALF program must contain exactly one main procedure, either in the program file or imported (see below). (Note that "un-runnable" ALF programs, or parts thereof, still may be perfectly analyzable.)

1.7 Assertions

Assertions can be encoded in ALF using the `switch` statement:

{ `switch` NUM_EXPR { `target` { `bin_val` 1 1 } $LABEL\_EXPR$ } }

If $LABEL\_EXPR$ is a label immediately succeeding the switch statement, then the switch will implement an assertion for NUM_EXPR. This is since implicit fall-through is disallowed for switches in ALF: thus, if NUM_EXPR evaluates to 0 the execution will fail, and if it evaluates to 1 then it will simply continue with the succeeding statement.

1.8 Scopes

ALF statements can be grouped into local environments called *scopes*. Scopes can have local variables (frames), which are allocated at entry and deallocated at exit. Scopes can be nested. If local variables with the same name are declared in nested scopes, then it is the closest declaration that is visible.

A scope can only be entered at its beginning: thus, it is not allowed to jump into a scope. It can be exited in the following ways:

- at its end,
- by a `return` statement, which returns from the most recent function call (see below), or
- by a `jump` statement, where the maximal number of exited, nested scopes is specified by n in the `leaving` n part of the statement. Thus, "`leaving` 1" specifies a jump out to (at most) the next scope level, and "`leaving` 0" specifies a jump within the same scope. Conditional jumps using the `switch` statement must have the selected target within the same scope. Any jump not adhering to these rules is considered a runtime error.

1.9 Functions

ALF has procedures, or *functions*. A function has a name and a list of formal arguments, which are frames. The body of a function is a scope, which means that a function also can have local variables. The formal arguments are similar to locally declared variables in scopes, with the difference that they are initialized with the values of the actual arguments when the function is called.

As explained in Section 1.6, a function is exited by a `return` statement. A function is also exited when the end of its body is reached, but the result parameters will then be undefined after the return of the call.

Functions can only be declared at a single, global level. and a declared function is visible in the whole ALF program. This is similar to C. Function names in ALF are labels with offset zero: calling a label with nonzero offset is considered a runtime error.

Addresses to functions can be passed as parameters, much as function pointers in C. This raises the question of *scoping rules* in ALF: when a function refers to a non-local variable, which one should that be? ALF has *static scoping*: the variable seen is the one seen from the function at its point of definition, not the point where it is called.

1.10 Type System

ALF has a simple, dynamic, monomorphic system with subtyping. All types but one are parameterized with respect to a (representation) *size*, which can be a natural number or infinity (meaning unlimited size). There are two classes of types (reflected in the system): *bitstring* types (mainly numerical), whose binary representation is fully known, and *symbolic* types (symbolic adresses) whose data have a symbolic contents. The type system has the following basic types:

- `size`: constant natural numbers plus the special constant `inf` which represents infinity.
- `anytype(n)`, where `n` is of type `size`, which is a supertype to all other types of size `n`.
- `bitstring(n)`, bitstring of length `n`. This is a supertype to all types whose data has a fully known binary representation, which for now are the limited size numerical types.
- `symbolic(n)`, supertype of all symbolic types of (assumed) size `n`. For now, the symbolic types are the different address types in ALF.
- `int(n)`, supertype for the numerical types which are bitstrings interpreted as integers.
- `unsigned(n)`, bitstring of length `n` interpreted as an unsigned number in the interval $[0, 2^n - 1]$.
- `signed(n)`, bitstring of length `n` interpreted as a signed number in the interval $[-2^{n-1}, 2^{n-1} - 1]$.
- `float(m,n)`, bitstring of length `m+n+1` interpreted as a floating-point IEEE number: `m` is the size of the fraction and `n` the size of the exponent.
- `fref(n)`, framerefs of size `n`.
- `address(n)`, data adresses in frames (frameref with offset) of size `n`.
- `lref(n)`, label references of size `n`.
- `label(n)`, code adresses (labels) of size `n`.

In addition, there are function types of form `(t1,...,tn) -> t` where `t` is a basic type different from `size`, and `t1,...,tn` are either basic types different from `size` or expressions of form `n:size` where `n` is an identifier. The latter construct binds `n` within the scope of the type signature, where it can be used in parameters to basic types. The meaning of `n:size` is that `n` is bound to the corresponding parameter. ALF only permits these parameters to be constants: thus, they will be known at type inference time.

Finite numerical types of size `n` stand for numerical values that are interpretations of bitstrings of length `n`. This means that operations on values of these types really are operations on bitstrings. This leads to the somewhat strange rule that all

subtypes to `bitstring(n)` are interchangeable: if a formal function argument has such a type, then the corresponding actual argument may be of any other such type. The type signature then merely serves as information about the intended interpretation of the argument (or result) for a given function. For instance, if

```
f : (unsigned(32),signed(32)) -> signed(32)
```

then `f`'s first argument can have any of the types `bitstring(32)`, `signed(32)`, or `float(m,n)` where $\mathtt{m}+\mathtt{n}+1 = 32$ as well. Since there are 1-1 mappings between all these types, the meaning is well-defined in all cases.

If the size equals `inf`, then the type has unbounded size. For instance, `unsigned(inf)` represents the set of natural numbers, and `signed(inf)` the integers. These types are useful if ALF is used to give semantics for instruction sets.

For the symbolic types, the size parameter gives an assumed size for the representation. This information is important when data of these types is stored in memory, to know how many bits will be occupied. For instance, `label(32)` is the type of 32-bit code addresses.

Most operators in ALF are defined for bitstring types only: this is natural, since the exact semantics for the operators in most cases requires that the representations of the operands are fully known. Symbolic address data can be loaded and stored. Some limited address arithmetic is also allowed: it remains to fit this into the type system, where arithmetic operators currently only can take operands of bitstring type.

As mentioned in Section 1.4, values of different types cannot in general be equal. Thus, $\mathtt{-1:signed(4)} \neq \mathtt{-1:signed(8)}$. This is sound since there are operations, like storing to memory, which have different effects for these values. However, values with bitstring types are considered equal iff their underlying bitstring representation is equal, even if they have different type. So, for instance, holds that $\mathtt{-1:signed(4)} = \mathtt{15:unsigned(4)}$.

As mentioned, ALF's current type system is dynamic. The reason is that the ability to represent low-level code makes it hard to assign types statically to memory locations. Indeed, a memory location may at different times contain values of different types. What could be done is to design a static type system that uses subtyping: such a type system would, for a given memory location, find a supertype that as tightly as possible describes the possible types of the values that might be stored in that location. Such a type system could be very useful for deciding the possible interpretations of stored values, which can be used to optimize value representations in program analyses. However, the definition of such a type system remains future work.

1.11 Namespaces, Import/Export, Scoping

Since ALF is supposed to support reverse engineering of binaries and object code, it cannot have an advanced module system with different namespaces. This is since most systems do not support object code with different name spaces, rather the linker will just look for imported names from any linked file. Thus, ALF has a simpler way of handling names through import and export statements:

The statement $\{$`export` $FREFS\ LREFS\}$ makes the framerefs and lrefs in $FREFS$ and $LREFS$ visible for other files being linked with the file, and $\{$`import` $FREFS\ LREFS\}$ requests that the identifiers and labels are exported from some other file being linked.

The "linker" in this case is a tool that is used to analyze whole programs with sources from different ALF files. It just performs a symbolic linking of names between files, not any mapping to memory addresses.

As mentioned in Section 1.9, ALF has static scoping. ALF has a global scope, and one local scope for each `scope` construct in the code. All imported entities are in the global scope, as well as framerefs declared in the "`decls`" section of an ALF program. As mentioned in Section 1.9, all functions are globally scoped. Indeed, all lrefs are globally scoped (although it is not allowed to jump to a label in `scope` from the outside, see Section 1.8). Framerefs that are locally declared have their scope restricted to their `scope` of declaration.

It is allowed to import an entity and declare it locally at the same time. The local declaration will then be visible. If the entity is exported, then it is the local declaration that is exported.

1.12 Macros

Macros can be declared in ALF. Macros have a name and can be given arguments: they can be seen as simple functions that are expanded at "compile-time". Macros can expand into most of ALF's syntactic forms, and can also take most such

forms as arguments. ALF macros do *not* expand into strings, as C #DEFINE's do, but into parse trees (or formal expressions). This means that ALF macros cannot be handled by a simple preprocessor, rather their implementation must be integrated into the parser. The grammar in Section 2 reflects this.

The minimal requirement on an ALF implementation is that macros are syntactically expanded into the parse trees they define. In addition, an ALF macro implementation may evaluate constant subexpressions during the macro expansion, which essentially turns the ALF macros into a simple functional language with support for recursion. But ALF implementation need not be able to handle recursive macros.

1.13 Annotations

Tools that analyze ALF might allow annotations that express external knowledge of relevance for the analysis. Such annotations might refer to ALF entities such as frames, or labels. However, the formats of such annotations are not part of the ALF language itself, and are thus not described here.

1.14 Semantics

ALF is an imperative language, with a relatively standard semantics based on state transitions. The state is comprised of the contents in data memory, a program counter (PC) holding the (possibly implicit) label of the current statement to be executed, and some representation of the stacked environments for (possibly recursive) function calls. A semiformal semantics, using a fairly concrete abstract machine, is given in Section 3.

Also programs which may jump to (or call) a non-existing label are allowed: they are called *partial ALF* programs. The semantics of a jump (call) to a non-existing label is that the program terminates in a state where the contents of the PC is the non-existing target label, and the rest of the state is as before the jump (call). The reason to give also partial ALF programs a semantics is that such programs may be useful when reverse-engineering binaries, where possible jump targets are iteratively determined using a program analysis.

The explicitly undefined value in ALF requires special consideration. For program analyses, its interpretation is TOP (no information). For the computation semantics, the result of applying an operation to this value depends on the strictness of the operation: for strict operations, the result is undefined if any argument is undefined, and for non-strict operations the results depend on whether the undefined argument(s) is needed to calculate the result. A load from an undefined address returns an undefined value. A store to an undefined address yields a memory contents where each address holds an undefined value. A jump to the undefined label yields a similar result as a jump to a non-existing label: termination with PC = the undefined label, and the rest of the state as immediately before the jump.

Addresses can also be partially undefined: the base address or the offset may be undefined, and the other part of the address defined. Loads and jumps treat partially undefined addresses in the same way as totally undefined adresses, but a store to a partially undefined address will affect only the part of the memory that is specified by the address. If the base address (frameref) is defined, then only the contents of the addresses in that frame will be set to undefined, and if the offset is defined then only the contents of the addresses with that offset will be set to undefined.

Accessing a smaller part of a stored undefined value (e.g., accessing eight bits out of the 32-bit undefined value) yields undefined.

2 Grammar

How to Read the Grammar. Words set in italics (usually capital letters) are non-terminals. Words in typewriter font are terminals, as are {, and }. Meta-symbols are "|" (or, union), "(", ")" (grouping), "?" (zero or one occurrence), " *" (Kleene star, zero or more occurrences), and "+" (one or more occurrences). The first element in a tuple is usually an operator, or a key for identifiying the type of the tuple (similar to data constructors on languages such as Haskell or ML). ALF uses an Erlang-style syntax, with { } for tuples (or structs) of fixed size.

The grammar does not explicitly specify whitespace. If (non-meta-)symbols are separated by a space in the grammar, then they are supposed to be separated by one or more whitespaces in the parsed code. Whitespace is either *SPACE* , *TAB* , or *RETURN* .

The grammar also does not specify comments. ALF has C-style comments: `/* .... */`, where the string contained within "`/*`" and "`*/`" can be any string not containing "`*/`". These comments can appear anywhere in the ALF code instead

of whitespace. The grammar below, however, assumes that there are no comments in the code. For instance, a pre-pass can strip them away.

How to Use the Grammar. The grammar cannot be used to define a parser right away. The reason is that there are redundancies in it, which will yield conflicts. These redundancies typically arise from different nonterminals which really are "synonyms" and have the same syntax: these nonterminals are usually then introduced to distinguish between semantically different entities with the same syntax. This is to help the reader, wishing to understand ALF, rather than the implementor. In the future we might publish a "reduced", conflict-free syntax derived from the one below.

The start symbol is *ALF*.

ALF → { `alf` *DEFS LAU ENDIAN EXPORTS IMPORTS DECLS INITS FUNCS* }

Macro definitions. Formal macro arguments start with "`@` " to distinguish them from other kinds of identifiers.

DEFS → { `macro_defs` *DEF* * }
DEF → { `def` { *DEF_ID FORMAL_ARG* * } *DEFINABLE* }
DEF_ID → *IDENTIFIER*
FORMAL_ARG → `@` *IDENTIFIER*

DEFINABLE is the set of all syntactic forms in the grammar that can be defined through a macro.

DEFINABLE → *FREF* | *LREF* | *FREF_ID* | *LREF_ID* | *ALLOC* | *INIT*
| *REF* | *OFFSET* | *VAL* | *REPEATS* | *CHAR_STRING* | *SCOPE*
| *STMTS* | *LABEL_STMT* | *LREF_EXPR* | *FREF_EXPR* | *ADDR_EXPR* | *NUM_EXPR*
| *CONST* | *LABEL* | *ADDR* | *NUM_VAL* | *INT_NUM_VAL* | *FLOAT_VAL*
| *HEX_STRING* | *SIGNED_VAL* | *UNSIGNED_STRING* | *BIN_STRING* | *SIZE_IN_BITS* | *SIZE_IN_LAU*

MACROCALL defines the syntax for macro calls to be expanded.

MACROCALL → { *DEF_ID ACTUAL* * }

ACTUAL is the set of all syntactic forms in the grammar that can appear as actual arguments to macros.

ACTUAL → *FREF* | *LREF* | *FREF_ID* | *LREF_ID*
| *OFFSET* | *VAL* | *REPEATS* | *CHAR_STRING* | *SCOPE*
| *STMTS* | *LABEL_STMT* | *LREF_EXPR* | *FREF_EXPR* | *ADDR_EXPR* | *NUM_EXPR*
| *CONST* | *LABEL* | *ADDR* | *NUM_VAL* | *INT_NUM_VAL* | *FLOAT_VAL*
| *HEX_STRING* | *SIGNED_VAL* | *UNSIGNED_STRING* | *BIN_STRING* | *SIZE_IN_BITS*

Least Adressable Unit on architecture.

LAU → { `least_addr_unit` *SIZE_IN_BITS* }

Endianness:

ENDIAN → `little_endian` | `big_endian`

Framerefs and (function) lrefs which are externally exported.

EXPORTS → { `exports` *FREFS LREFS* }
FREFS → { `frefs` *FREF* * }
LREFS → { `lrefs` *LREF* * }

Framerefs and (function) lrefs that needs to be imported.

IMPORTS → { `imports` *FREFS LREFS* }

Framerefs and lrefs are essentially symbols. (Note the cases *MACROCALL* and *FORMAL_ARG* : the first is because a macro can return a *FREF* , *LREF* , *FREF_ID* , or *LREF_ID* , and the second is because any of these can appear as actual argument to a macro and thus may be represented by a formal argument in the macro body. Each syntactic category that can

be defined as a macro has a *MACROCALL* in its production rule, and each that can be represented by a formal argument to a macro has *FORMAL_ARG* in its rule.)

FREF → { `fref` *SIZE_IN_BITS FREF_ID* } | *MACROCALL* | *FORMAL_ARG*
LREF → { `lref` *SIZE_IN_BITS LREF_ID* } | *MACROCALL* | *FORMAL_ARG*
FREF_ID → *IDENTIFIER* | *MACROCALL* | *FORMAL_ARG*
LREF_ID → *IDENTIFIER* | *MACROCALL* | *FORMAL_ARG*

Allocation of global data.

DECLS → { `decls` *ALLOC* * }

Static allocation of a frame of a certain *SIZE_IN_BITS* (in bits), with *FREF_ID* as the symbolic part of its frameref (at most one alloc per *FREF_ID* is allowed in *GLOB_DECLS*). The first *SIZE_IN_BITS* is the size of the bitstring representation of the corresponding frameref.

ALLOC → { `alloc` *SIZE_IN_BITS FREF_ID SIZE_IN_BITS* } | *MACROCALL*

Initialization of data. Initialized data can also be declared as volatile or read-only (but not both).

INITS → { `inits` *INIT* * }

To initialize the data referenced with value.

INIT → { `init` *REF VAL* (`volatile` | `read_only`)? } | *MACROCALL*

The initialization is made of data located from the address corresponding to the frameref named *FREF_ID* + offset and up. If the initialized data is declared volatile or read-only, then that applies to all addresses that are actually initialized.

REF → { `ref` *FREF_ID OFFSET* } | *MACROCALL*

Offset should be a (constant) natural number.

OFFSET → *INT_NUM_VAL* | *MACROCALL* | *FORMAL_ARG*

Each constant holds its own size. *CONST* is a single constant value. `const_repeat` repeats a constant a number of times. `const_list` is a list of consecutive constants of same type and size.

VAL → *CONST*
| { `const_repeat` *CONST REPEATS* }
| { `const_list` *CONST* + }
| { `hex_list` *SIZE_IN_BITS HEX_STRING* + }
| { `dec_list` *SIZE_IN_BITS SIGNED_VAL* + }
| { `udec_list` *SIZE_IN_BITS UNSIGNED_STRING* + }
| { `bin_list` *SIZE_IN_BITS BIN_STRING* + }
| { `float_list` *SIZE_IN_BITS SIZE_IN_BITS FLOAT_VAL* + }
| { `char_string` *CHAR_STRING* }
| *MACROCALL* | *FORMAL_ARG*

The number of times a constant is repeated should be a natural number:

REPEATS → *UNSIGNED_STRING* | *MACROCALL* | *FORMAL_ARG*

8-bit character string, delimited by `" "`. It uses C string literal syntax (represented by the nonterminal "*C_STRING_LITERAL*"), which means exactly the same escape codes are allowed as in C string literals.

CHAR_STRING → *C_STRING_LITERAL* | *MACROCALL* | *FORMAL_ARG*

Functions.

FUNCS → { `funcs` *FUNC* * }

Each function declaration has a label, which provides its name. The label must have offset zero. Frames allocated in ARG_DECLS hold actual arguments, after evaluation (call by value). The arguments are provided in a `call` statement.

FUNC → { `func` *LABEL ARG_DECLS SCOPE* }
ARG_DECLS → { `arg_decls` *ALLOC* * }

A scope is a block which can contain locally defined variables (frames), and code. ALF has static scoping. ALF offers the possibility to initialize frames, which are dynamically allocated in a scope, through an init declaration: if a faithful modelling of the program flow is required then this mechanism should be avoided, since in reality some code will have to execute in order to initialize the data.

SCOPE → { `scope` *DECLS INITS STMTS* } | *MACROCALL* | *FORMAL_ARG*

ALF code is an ordered list of possibly labelled statements. There is no explicit CFG representation in ALF.

STMTS → { `stmts` *LABEL_STMT* * }

A statement may have a label (but does not have to).

LABEL_STMT → *LABEL STMT*
| *STMT*
| *MACROCALL* | *FORMAL_ARG*

Statements. (See Section 1.6.)

STMT → { `null` }
| { `store` *ADDR_EXPR* + `with` *EXPR* + }
| { `jump` *LABEL_EXPR* `leaving` *UNSIGNED_STRING* }
| { `switch` *NUM_EXPR TARGET* + }
| { `call` *LABEL_EXPR EXPR* * `result` *ADDR_EXPR* * }
| { `return` *EXPR* * }
| { `free` *FREF_EXPR* }
| *SCOPE*
| *MACROCALL* | *FORMAL_ARG*

TARGET → { `target` *INT_NUM_VAL LABEL_EXPR* }
| { `default` *LABEL_EXPR* }

The ALF grammar defines five different kinds of expressions: *LREF_EXPR* , *LABEL_EXPR* , *FREF_EXPR* , *ADDR_EXPR* , and *NUM_EXPR* . This is mainly for showing that there is an underlying concept of different expression types. However, ALF cannot be given a fully static type system: the main culprit is the `load` operation, which in general will be able to return a value of any type. Thus, the rules for the different kinds of expressions all contain a call to `load`, which renders the grammar ambiguous the way it stands. The solution, when implementing a parser for the grammar, is to merge the five different expression types into a single syntactic category *EXPR* .

A $LREF\_EXPR$ should evaluate to an LREF (symbolic base pointer for label). (`load` loads a value from the address given by ADDR_EXPR, and { `undefined` $SIZE\_IN\_BITS$ } represents an unknown value of size *SIZE_IN_BITS*.)

LREF_EXPR → *LREF*
| { `load` *SIZE_IN_BITS ADDR_EXPR* }
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

A $LABEL\_EXPR$ expression should evaluate to a LABEL (of function or stmt). It can be a (constant) label, a *computed* label, a label loaded from memory, or undefined. Limited address arithmetic is also allowed (addition, subtraction with numerical value: the numerical value is then added to (subtracted from) the offset of the label).

LABEL_EXPR → *LABEL*
| { `label` *SIZE_IN_BITS LREF_EXPR NUM_EXPR* }
| { `load` *SIZE_IN_BITS ADDR_EXPR* }
| { `add` *SIZE_IN_BITS LABEL_EXPR NUM_EXPR NUM_EXPR* }
| { `add` *SIZE_IN_BITS NUM_EXPR LABEL_EXPR NUM_EXPR* }
| { `sub` *SIZE_IN_BITS LABEL_EXPR NUM_EXPR NUM_EXPR* }
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

A $FREF\_EXPR$ expression evaluates to a frameref (identifying a frame). `dyn_alloc` allocates a frame dynamically, with size *NUM_EXPR* (counted in LAU) and representation of size *SIZE_IN_BITS* (in bits), and it returns a frameref.

FREF_EXPR → *FREF*
| { `dyn_alloc` *SIZE_IN_BITS FREF_ID NUM_EXPR* }
| { `load` *SIZE_IN_BITS ADDR_EXPR* }
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

An $ADDR\_EXPR$ evaluates to an address (within a frame), expressed as a frameref and an offset in LAU from the start of the frame. *ADDR* is a special form for constant addresses, with simplified syntax. Limited address arithmetic is allowed (addition, subtraction with numerical value: the numerical value is then added to (subtracted from) the offset of the address). All NUM_EXPRs define offsets, or offset increments or decrements, in LAU.

ADDR_EXPR → *ADDR*
| { `addr` *SIZE_IN_BITS FREF_EXPR NUM_EXPR* }
| { `load` *SIZE_IN_BITS ADDR_EXPR* }
| { `add` *SIZE_IN_BITS ADDR_EXPR NUM_EXPR NUM_EXPR* }
| { `add` *SIZE_IN_BITS NUM_EXPR ADDR_EXPR NUM_EXPR* }
| { `sub` *SIZE_IN_BITS ADDR_EXPR NUM_EXPR NUM_EXPR* }
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

A NUM_EXPR expression should evaluate to a numerical value. *OP* is a builtin operator (function) with a predefined # of arguments. Some operators also take a certain number of *size arguments* which are integer constants defining the representation sizes (in bits) of arguments and/or results. For `sub`, both $ADDR_EXPR$s should evaluate to addresses within the same frame.

NUM_EXPR → *NUM_VAL*
| { `load` *SIZE_IN_BITS ADDR_EXPR* }
| { *OP SIZE_IN_BITS* NUM_EXPR** }
| { `sub` *SIZE_IN_BITS ADDR_EXPR ADDR_EXPR NUM_EXPR* }
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

An expression $EXPR$ could evaluate to an lref, a label, a frameref, an address, or a numerical value.

EXPR → *LREF_EXPR*
| *LABEL_EXPR*
| *FREF_EXPR*
| *ADDR_EXPR*
| *NUM_EXPR*

Forms for specifying constant values of different types (used, for instance, in initialization of data).

CONST → *LREF*
| *ADDR*
| *FREF*
| *LABEL*
| *NUM_VAL*
| { `undefined` *SIZE_IN_BITS* }
| *MACROCALL* | *FORMAL_ARG*

LABEL → { `label` *SIZE_IN_BITS LREF OFFSET* } | *MACROCALL* | *FORMAL_ARG*
ADDR → { `addr` *SIZE_IN_BITS FREF OFFSET* } | *MACROCALL* | *FORMAL_ARG*

Identifiers can be of two forms. In the first form, identifiers are composed from alphanumerical characters plus a few special characters. They must then start with an alphabetic character, and they are case-sensitive. Identifiers of this form must not be the same as reserved keywords in ALF (essentially the terminals in this grammar), but this is not reflected in the grammar. In the second form, identifiers are C string literals.

Two identifiers represented by C string literals are equal exactly when the corresponding byte arrays are equal. An identifier of the first form is equal to an identfier of the second form which is the first identifier within quotes (so, for instance, `Aaa` is same identifier as `"Aaa"`).

IDENTIFIER → *ALPHA* (*ALPHA* | *NUM* | *SPECIAL*)* | *C_STRING_LITERAL*
ALPHA → `a` | ... | `z` | `A` | ... | `Z`
NUM → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
SPECIAL → `_` | `.` | `%` | `$` | `@` | `:` | `'`

A numerical constant may have many different types: `signed(n)`, `unsigned(n)`, `bitstring(n)`, etc. This is reflected in the syntax for numerical constants. However, no matter which syntax is chosen the resulting underlying bitstring should be uniquely defined both as regards length and contents.

The two sizes for `float_val` below are for floating-point exponent and fraction, respectively. For IEEE single precision 32-bit floating point numbers the values of these are 8 and 23, respectively, and for double precision 64-bit they are 11 and 52. Since there is also a sign bit, the total size is $exp + frac + 1$ where exp and $frac$ are the two specified sizes.

NUM_VAL → *INT_NUM_VAL*
| { `float_val` *SIZE_IN_BITS SIZE_IN_BITS FLOAT_VAL* }
| *MACROCALL* | *FORMAL_ARG*

INT_NUM_VAL → { `hex_val` *SIZE_IN_BITS HEX_STRING* }
| { `dec_signed` *SIZE_IN_BITS SIGNED_VAL* }
| { `dec_unsigned` *SIZE_IN_BITS UNSIGNED_STRING* }
| { `bin_val` *SIZE_IN_BITS BIN_STRING* }
| *MACROCALL* | *FORMAL_ARG*

HEX_STRING → *H* + | *MACROCALL* | *FORMAL_ARG*

SIGNED_VAL → *UNSIGNED_STRING*
| { `minus` *UNSIGNED_STRING* }
| *MACROCALL* | *FORMAL_ARG*

UNSIGNED_STRING → *NUM* + | *MACROCALL* | *FORMAL_ARG*

BIN_STRING → (`0` | `1`)+ | *MACROCALL* | *FORMAL_ARG*

FLOAT_VAL is C syntax for floating-point numbers, except the possible specification of long/double format at end (which is overridden by ALF's more precise size specification).

FLOAT_VAL → −?(*NUM* + `.` *NUM* * | `.` *NUM* +)((`e` | `E`)(`+` | `−`)?*NUM* +)? | (*NUM* +(`e` | `E`)(`+` | `−`)?*NUM* +)
| *MACROCALL* | *FORMAL_ARG*

SIZE_IN_LAU → *UNSIGNED_STRING*
| `inf`
| *MACROCALL* | *FORMAL_ARG*

SIZE_IN_BITS is usually 8, 16, 32 or 64, but can be any other positive number and even `inf`, meaning unbounded size.

SIZE_IN_BITS → *UNSIGNED_STRING*
| `inf`
| *MACROCALL* | *FORMAL_ARG*

ALF's different operations are split into six subgroups: *OP_INT* , *OP_BIT* , *OP_FLOAT* , *OP_CMP* , *OP_MATH* , and *OP_BITSTR* . This is mostly to obtain a nicer-looking grammar, and an implementation can well merge these directly into a single syntactic category *OP* .

OP → *OP_INT* | *OP_BIT* | *OP_FLOAT* | *OP_CMP* | *OP_MATH* | *OP_BITSTR*

For ALF's different operations, we also give the respective type signature as well as a comment on the meaning of the operation. The types and comments are interleaved in the grammar, but strictly not part of it. Function types of form "`(m:size,n:size,...) -> ...`" are abbreviated as "`(m,n:size,...) -> ...`".

OP_INT →
- `neg`
  $(\mathtt{n} : \mathtt{size}, \mathtt{signed}(\mathtt{n})) \to \mathtt{signed}(\mathtt{n})$
  (Signed) integer negation
- | `add`
  $(\mathtt{n} : \mathtt{size}, \mathtt{int}(\mathtt{n}), \mathtt{int}(\mathtt{n}), \mathtt{unsigned}(1)) \to \mathtt{int}(\mathtt{n})$
  integer addition (with carry, set carry $=$ 0 (third argument) to represent add without carry-in)
- | `c_add`
  $(\mathtt{n} : \mathtt{size}, \mathtt{int}(\mathtt{n}), \mathtt{int}(\mathtt{n}), \mathtt{unsigned}(1)) \to \mathtt{unsigned}(1)$
  Carry out from integer addition
- | `sub`
  $(\mathtt{n} : \mathtt{size}, \mathtt{int}(\mathtt{n}), \mathtt{int}(\mathtt{n}), \mathtt{unsigned}(1)) \to \mathtt{int}(\mathtt{n})$
  Integer subtraction with carry (not borrow, set borrow $=$ 1 (third argument) to represent sub without carry-in)
- | `c_sub`
  $(\mathtt{n} : \mathtt{size}, \mathtt{int}(\mathtt{n}), \mathtt{int}(\mathtt{n}), \mathtt{unsigned}(1)) \to \mathtt{unsigned}(1)$
  Carry out (not borrow) from integer subtraction : set exactly when first operand $\geq$ second operand (seen as unsigned)
- | `u_mul`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{unsigned}(\mathtt{m}), \mathtt{unsigned}(\mathtt{n})) \to \mathtt{unsigned}(\mathtt{m} + \mathtt{n})$
  Unsigned integer multiplication
- | `s_mul`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{signed}(\mathtt{m}), \mathtt{signed}(\mathtt{n})) \to \mathtt{signed}(\mathtt{m} + \mathtt{n})$
  Signed integer multiplication
- | `u_div`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{unsigned}(\mathtt{m}), \mathtt{unsigned}(\mathtt{n})) \to \mathtt{unsigned}(\mathtt{m})$
  Unsigned integer division
- | `s_div`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{signed}(\mathtt{m}), \mathtt{signed}(\mathtt{n})) \to \mathtt{signed}(\mathtt{m})$
  Signed integer division
- | `u_mod`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{unsigned}(\mathtt{m}), \mathtt{unsigned}(\mathtt{n})) \to \mathtt{unsigned}(\mathtt{m})$
  Unsigned integer remainder
- | `s_mod`
  $(\mathtt{m}, \mathtt{n} : \mathtt{size}, \mathtt{signed}(\mathtt{m}), \mathtt{signed}(\mathtt{n})) \to \mathtt{signed}(\mathtt{m})$
  Signed integer remainder

OP_BIT → `l_shift`
(`n`, `m` : `size`, `bitstring`(`n`), `unsigned`(`m`)) → `bitstring`(`n`)
(Logical) left shift (2nd arg gives no of positions to shift)

| `r_shift`
(`n`, `m` : `size`, `bitstring`(`n`), `unsigned`(`m`)) → `bitstring`(`n`)
(Logical) right shift

| `r_shift_a`
(`n`, `m` : `size`, `bitstring`(`n`), `unsigned`(`m`)) → `bitstring`(`n`)
Arithmetical right shift

| `s_ext`
(`m`, `n` : `size`, `bitstring`(`m`)) → `bitstring`(`n`)
Sign extend ($m \leq n$)

| `not`
(`n` : `size`, `bitstring`(`n`)) → `bitstring`(`n`)
Bitwise NOT

| `and`
(`n` : `size`, `bitstring`(`n`), `bitstring`(`n`)) → `bitstring`(`n`)
Bitwise AND

| `or`
(`n` : `size`, `bitstring`(`n`), `bitstring`(`n`)) → `bitstring`(`n`)
Bitwise OR

| `xor`
(`n` : `size`, `bitstring`(`n`), `bitstring`(`n`)) → `bitstring`(`n`)
Bitwise XOR

OP_FLOAT → `f_neg`
(`m`, `n` : `size`, `float`(`m`, `n`)) → `float`(`m`, `n`)
Floating point negation

| `f_add`
(`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `float`(`m`, `n`)
Floating point addition

| `f_sub`
(`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `float`(`m`, `n`)
Floating point subtraction

| `f_mul`
(`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `float`(`m`, `n`)
Floating point multiplication

| `f_div`
(`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `float`(`m`, `n`)
Floating point division

| `f_to_f`
(`m1`, `m2`, `n1`, `n2` : `size`, `float`(`m1`, `n1`)) → `float`(`m2`, `n2`)
Floating point to floating point cast

| `f_to_u`
(`m1`, `n1`, `n` : `size`, `float`(`m1`, `n1`))) → `unsigned`(`n`)
Floating point to unsigned integer cast

| `f_to_s`
(`m1`, `n1`, `n` : `size`, `float`(`m1`, `n1`)) → `signed`(`n`)
Floating point to signed integer cast

| `u_to_f`
(`m1`, `n1`, `m` : `size`, `unsigned`(`m`)) → `float`(`m1`, `n1`)
Unsigned integer to floating point cast

| `s_to_f`
(`m1`, `n1`, `m` : `size`, `signed`(`m`)) → `float`(`m1`, `n1`)
Signed integer to floating point cast

OP_CMP →

| `eq`
  (`n` : `size`, `anytype`(`n`), `anytype`(`n`)) → `unsigned`(`1`)
  Compare for (integer) equality
| `neq`
  (`n` : `size`, `anytype`(`n`), `anytype`(`n`)) → `unsigned`(`1`)
  Compare for (integer) inequality
| `u_lt`
  (`n` : `size`, `unsigned`(`n`), `unsigned`(`n`)) → `unsigned`(`1`)
  Unsigned integer compare less than
| `u_ge`
  (`n` : `size`, `unsigned`(`n`), `unsigned`(`n`)) → `unsigned`(`1`)
  Unsigned integer compare greater than or equal
| `u_gt`
  (`n` : `size`, `unsigned`(`n`), `unsigned`(`n`)) → `unsigned`(`1`)
  Unsigned integer compare greater than
| `u_le`
  (`n` : `size`, `unsigned`(`n`), `unsigned`(`n`)) → `unsigned`(`1`)
  Unsigned integer compare less than or equal
| `s_lt`
  (`n` : `size`, `signed`(`n`), `signed`(`n`)) → `unsigned`(`1`)
  Signed integer compare less than
| `s_ge`
  (`n` : `size`, `signed`(`n`), `signed`(`n`)) → `unsigned`(`1`)
  Signed integer compare greater than or equal
| `s_gt`
  (`n` : `size`, `signed`(`n`), `signed`(`n`)) → `unsigned`(`1`)
  Signed integer compare greater than
| `s_le`
  (`n` : `size`, `signed`(`n`), `signed`(`n`)) → `unsigned`(`1`)
  Signed integer compare less than or equal
| `f_eq`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare equal
| `f_ne`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare not equal
| `f_lt`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare less than
| `f_ge`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare greater than or equal
| `f_gt`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare greater than
| `f_le`
  (`m`, `n` : `size`, `float`(`m`, `n`), `float`(`m`, `n`)) → `unsigned`(`1`)
  Floating point compare less than or equal

OP_MATH →

| `if`
  (`n` : `size`, `unsigned`(`1`), `anytype`(`n`), `anytype`(`n`)) → `anytype`(`n`)
  Conditional
| `b2n`
  `bitstring`(`n`) → `unsigned`(`inf`)
  Conversion bitstring to natural number
| `exp2`
  `unsigned`(`inf`) → `unsigned`(`inf`)
  Two − exponent : { `exp2` n } = 2^{n}

OP_BITSTR →

| `select`
  (`k`, `m`, `n` : `size`, `bitstring`(`k`)) → `bitstring`(`n` − `m` + `1`)
  Substring of bitstring from `m` to `n` ($0 \leq \mathtt{m} \leq \mathtt{n} < \mathtt{k}$). Bits are numbered from less to more significant, starting with 0
| `conc`
  (`m`, `n` : `size`, `bitstring`(`m`), `bitstring`(`n`)) → `bitstring`(`m` + `n`)
  Concatenation of bitstrings. The first argument will occupy the more significant bits
| `repeat`
  (`n` : `size`, `unsigned`(`1`)) → `bitstring`(`n`)
  Bitstring with bit repeated `n` times

2.1 Reserved ALF Keywords

These are the reserved keywords in ALF (more or less the terminals in the grammar):

`add addr alf alloc and arg_decls b2n big_endian bin_list bin_val c_add call char_string conc const_list const_repeat c_sub dec_list decls dec_signed dec_unsigned def default dyn_alloc eq exp2 exports f_add f_div f_eq f_ge f_gt f_le float_list float_val f_lt f_mul f_ne f_neg free fref frefs f_sub f_to_f f_to_s f_to_u func funcs hex_list hex_val if imports inf init inits jump label least_addr_unit leaving little_endian load lref lrefs l_shift macro_defs minus neg neq not null or read_only ref repeat result return r_shift r_shift_a scope s_div select s_ext s_ge s_gt s_le s_lt s_mod s_mul stmts s_to_f store sub switch target udec_list u_div u_ge u_gt u_le u_lt u_mod u_mul undefined u_to_f volatile with xor`

2.2 Some Examples How To Model Operations in ALF

As a first example, we show how to model a 32-bit `add` operation (no carry in), which adds the contents of registers `r1` and `r2` and stores the result in `r0`. We model the registers as the three first elements (offsets 0, 1, 2) in a statically allocated frame named `r`:

```
{ store { addr 32 { fref 32 r } { dec_unsigned 32 0 }}
        { add 32 { load 32 { addr 32 { fref 32 r } { dec_unsigned 32 1 }}}
                 { load 32 { addr 32 { fref 32 r } { dec_unsigned 32 2 }}}
                 { dec_unsigned 1 0 }}}
```

The above example shows that ALF expressions can be quite unwieldy, since all details about data sizes etc. always must be specified explicitly in ALF. ALF programs can however be made more readable by the use of macros, and an ALF generator can put a well-selected preamble of macros in its generated ALF file for subsquent use. As an example, one can introduce some macros for avoiding having to specify bitlengths of data and operations. For instance, If data and operations are 32-bit, then we can make the following macro definitions:

```
{ def add32 { m n }  { add 32 m n { dec_unsigned 1 0 } }
{ def addr32 { f n } { addr 32 { fref 32 f } { dec_unsigned 32 n } } }
```

With these macros, the example above can be rewritten as:

```
{ store { addr32 r 0 }
        { add32 { load 32 { addr32 r 1 }}
                { load 32 { addr32 r 2 }}}}
```

As an example how to use the more "mathematical" operations of ALF to model the semantics of instructions, we specify the carry-out function for n-bit addition of two bitstrings x and y. Mathematically, this can be expressed as

$$b2n(x) + b2n(y) \div 2^n > 0$$

or, in ALF format:

```
{ gt inf { div inf inf { add inf { b2n x } { b2n y } { dec_unsigned inf 0 }}
                       { exp2 n}}
         { dec_unsigned inf 0 }}
```

3 Semantics for ALF

A note to the reader: this section is not up to date, it refers to an earlier version of ALF. However, ALF has not undergone any major changes since the section was written so although details may be off, the semantics still provides some insight in how ALF works.

3.1 Type system used in the semantics

The type system that is used in the semantics includes an extended variant of the type system used in ALF. While the type system in ALF is monomorphic, the type system used in the semantics is polymorphic. This means that, e.g., for a value `b` of type `bitstring(n)`, the expression `isFloat(b)` reveals whether or not it is representing a float value with `n = m + n'`, where `m` and `n'` are the sizes of the fraction and exponent, respectively.

In order to find the fref that corresponds to an identifier, environment functions are used that map identifiers to the right frefs.

3.2 About the semantic functions

The semantic functions all take syntax trees as input, and are expressed using imperative pseudo code. For simplicity, some parts of the tree are assumed to have already been parsed and turned into semantic values, and are stored as such in the tree. For example, a subtree resulting from the grammatic rule

INT_NUM_VAL → { `hex_val` *SIZE_IN_BITS* *HEX_STRING* }

is assumed to have already been turned into a value of type `unsigned(n)` with `n` equaling the value of *SIZE_IN_BITS* . Table 1 shows which parts of the syntax tree (i.e., rules of the grammar) are assumed to be stored as the values they represent. Furthermore, all macro calls are assumed to have been expanded in the syntax tree.

For lists, a syntax similar to Erlang is used, where `[h|t]` denotes a list with head `h` and tail `t`. This can be used to split a list and assign the head and tail to different variables. For instance, `[Head | Tail] <- [1, 2, 3, 4]` assigns `1` to the variable `Head` and `[2, 3, 4]` to the variable `Tail`. Also, the double-plus operator, `++`, applied to two lists results in a list that is the concatenation of them, e.g., `[1, 2] ++ [3, 4] = [1, 2, 3, 4]`.

For the environment functions, the *least upper bound* operator, `|_|`, is used to introduce new mappings in the environment. For example, if `Env` is some environment, then `Env |_| {(Symbol, FRef)}` is an environment which is the same as `Env` but with the additional mapping from the symbol `Symbol` to the fref `FRef`. Furthermore, `defined(Env, Symbol)` tells whether the environment `Env` has a mapping from the identifier `Symbol` to some fref.

Throughout the code, `assert` statements are used to express certain constraints that must hold for the ALF program to be correct.

Rule	Type
SIZE_IN_BITS	`size`
SIZE_IN_LAU	`size`
OFFSET	`signed(n)` or `unsigned(n)`
CONST	Type depends on the production
REPEATS	`unsigned(inf)`
HEX_STRING	`unsigned(inf)`
SIGNED_VAL	`signed(inf)`
UNSIGNED_STRING	`unsigned(inf)`
BIN_STRING	`bitstring(inf)`
CHAR_STRING	`unsigned(inf)`
FLOAT_VAL	`float(inf, inf)`
NUM_VAL	Type depends on the production
LREF	`lref(n)`
IDENTIFIER	`identifier`

Table 1. Grammar rules for the parts of the syntax tree that the semantics assume are stored as constants. The right column shows the types of the constants.

3.3 Semantic functions

3.3.1 Expressions

Evaluate expression Expr. Returns a value of type `anytype(n)` for some $n \in \texttt{size} = \mathbb{N} \cup \{\texttt{inf}\}$.

```
evalExpr(Expr, Env, CallStack, GEnv, Mem)
    // Choose an evaluation function based on the type of the expression
    if isNumExpr(Expr) then
        return evalNumExpr(Expr, Env, CallStack, GEnv, Mem)
    else
        return evalSymbExpr(Expr, Env, CallStack, GEnv, Mem)
    end if
end evalExpr
```

Evaluate numerical expression NumExpr. Returns a value of type `bitstring(n)`.

```
evalNumExpr(NumExpr, Env, CallStack, GEnv, Mem)
    // Handle the case of a simple expression, which corresponds to the NUM_VAL
    // and { undefined SizeInBits } rules in the grammar
    if isNumVal(NumExpr) then
        return NumExpr
    end if

    // Handle the remaining cases
    case NumExpr of
    { load SizeInBits AddrExpr }:
        // Evaluate the address expression and use it to load a value from
        // memory
        Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
        Value <- load(Mem, Addr, SizeInBits)
        assert(isNumVal(Value))
        assert(size(Value) = SizeInBits)
        return Value
```

```
    { Op SizeInBitList NumExprList }:
        return evalOpExpr(Op, SizeInBitList, NumExprList, Env, CallStack,
                          GEnv, Mem)

    { sub SizeInBits AddrExpr1 AddrExpr2 NumExpr2 }:
        // Take the difference between the offsets of the two addresses (which
        // are assumed to go to the same frame). The numerical expression
        // is the carry in to the subtraction.
        Addr1 <- evalAddrExpr(AddrExpr1, Env, CallStack, GEnv, Mem)
        Addr2 <- evalAddrExpr(AddrExpr2, Env, CallStack, GEnv, Mem)
        assert(fref(Addr1) = fref(Addr2))
        Addr1Offs <- offset(Addr1)
        Addr2Offs <- offset(Addr2)
        CarryIn <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        Diff <- subWCarry(Addr1Offs, Addr2Offs, CarryIn)
        assert(size(Diff) = SizeInBits)
        return Diff

    end case
end evalNumExpr
```

Evaluate symbolic expression SymbExpr. Returns a value of type `symbolic(n)`.

```
evalSymbExpr(SymbExpr, Env, GEnv, Mem)
    // Delegate to an appropriate semantic function
    if isLRefExpr(SymbExpr) then
        return evalLRefExpr(SymbExpr, Env, CallStack, GEnv, Mem)
    else if isLabelExpr(SymbExpr) then
        return evalLabelExpr(SymbExpr, Env, CallStack, GEnv, Mem)
    else if isFRefExpr(SymbExpr) then
        return evalFRefExpr(SymbExpr, Env, CallStack, GEnv, Mem)
    else if isAddrExpr(SymbExpr) then
        return evalAddrExpr(SymbExpr, Env, CallStack, GEnv, Mem)
    end if
end evalSymbExpr
```

Evaluate lref expression LRefExpr. Returns a value of type `lref(n)`.

```
evalLRefExpr(LRefExpr, Env, CallStack, GEnv, Mem)
    // Handle the simple case of LRefExpr being an LRef (this also handles the
    // undefined case)
    if isLRef(LRefExpr) then
        return LRefExpr
    end if

    // Handle the remaining case - evaluate address expression and use it to
    // load an lref from memory
    { load SizeInBits AddrExpr } <- LRefExpr
    Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
    LRef <- load(Mem, Addr, SizeInBits)
    assert(isLRef(LRef))
    assert(size(LRef) = SizeInBits)
    return LRef
```

```
end evalLRefExpr
```

Evaluate label expression LExpr. Returns a value of type `label(n)`.

```
evalLabelExpr(LExpr, Env, CallStack, GEnv, Mem)
    case LExpr of

    // The LABEL case has been merged into to this case
    { label SizeInBits LRefExpr NumExpr }:
        // Create and return a label
        LRef <- evalLRefExpr(LRefExpr, Env, CallStack, GEnv, Mem)
        Offset <- evalNumExpr(NumExpr, Env, CallStack, GEnv, Mem)
        return label(LRef, Offset, SizeInBits)

    { load SizeInBits AddrExpr }:
        // Evaluate address expression and load a label from memory
        Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
        Label <- load(Mem, Addr, SizeInBits)
        assert(isLabel(Label))
        assert(size(Label) = SizeInBits)
        return Label

    { add SizeInBits LExpr2 NumExpr1 NumExpr2 }:
        // Evaluate and add NumExpr1 to the offset of the label LExpr2
        Label <- evalLabelExpr(LExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Label)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- addWCarry(Offset, Value, Carry)
        return label(lref(Label), NewOffset, SizeInBits)

    { add SizeInBits NumExpr1 LExpr2 NumExpr2 }:
        // A variant of the previous operation
        Label <- evalLabelExpr(LExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Label)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- addWCarry(Offset, Value, Carry)
        return label(lref(Label), NewOffset, SizeInBits)

    { sub SizeInBits LExpr2 NumExpr1 NumExpr2 }:
        // Subtract NumExpr1 from the offset of the label LExpr2
        Label <- evalLabelExpr(LExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Label)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- subWCarry(Offset, Value, Carry)
        return label(lref(Label), NewOffset, SizeInBits)

    { undefined SizeInBits }:
        return undefLabel(SizeInBits)

    end case
```

```
end evalLabelExpr
```

Evaluate fref expression FRefExpr. Returns a value of type `fref(n)`.

```
evalFRefExpr(FRefExpr, Env, CallStack, GEnv, Mem)
    case FRefExpr of
    // The FREF production has been "inlined" here
    { fref SizeInBits FRefId }:
        // Find the fref corresponding to FRefId and return it
        return lookup(FRefId, Env, CallStack, GEnv)

    { dyn_alloc SizeInBits FRefId NumExpr }:
        // Create a new globally unique fref, then dynamically allocate a frame
        // of the size given by NumExpr which can be referenced by the newly
        // created fref. Finally return the fref. (Note that since no mapping is
        // introduced in any of the environments from FRefId to the fref, the
        // the fref cannot be accessed in the code through its identifier.)
        CounterVal <- readCounter(FRefId)
        incrCounter(FRefId)
        FRef <- fref(frefId(FRefId, CounterVal), SizeInBits)
        AllocSize <- evalNumExpr(NumExpr, Env, CallStack, GEnv, Mem)
        alloc(AllocSize, FRef)
        return FRef

    { load SizeInBits AddrExpr }:
        // Evaluate AddrExpr and load an fref from memory
        Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
        FRef <- load(Mem, Addr, SizeInBits)
        assert(isFRef(FRef))
        assert(size(FRef) = SizeInBits)
        return FRef

    { undefined SizeInBits }:
        return undefFRef(SizeInBits)

    end case
end evalFRefExpr
```

Evaluate address expression AddrExpr. Returns a value of type `address(n)`.

```
evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
    case AddrExpr of

    // The ADDR rule has been merged into this case
    { addr SizeInBits FRefExpr NumExpr }:
        // Create and return an address
        FRef <- evalFRefExpr(FRefExpr, Env, CallStack, GEnv, Mem)
        Offset <- evalNumExpr(NumExpr, Env, CallStack, GEnv, Mem)
        return addr(FRef, Offset, SizeInBits)

    { load SizeInBits AddrExpr2 }:
        // Evaluate AddrExpr2 into an address and use it to load an address from
        // memory
        LoadAddr <- evalAddrExpr(AddrExpr2, Env, CallStack, GEnv, Mem)
```

```
        Addr <- load(Mem, LoadAddr, SizeInBits)
        assert(isAddr(Addr))
        assert(size(Addr) = SizeInBits)
        return Addr

    { add SizeInBits AddrExpr2 NumExpr1 NumExpr2 }:
        // Evaluate and add NumExpr1 to the offset of the address LExpr2
        Addr <- evalAddrExpr(AddrExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Addr)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- addWCarry(Offset, Value, Carry)
        return addr(fref(Addr), NewOffset, SizeInBits)

    { add SizeInBits NumExpr1 AddrExpr2 NumExpr2 }:
        // A variant of the previous operation
        Addr <- evalAddrExpr(AddrExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Addr)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- addWCarry(Offset, Value, Carry)
        return addr(fref(Addr), NewOffset, SizeInBits)

    { sub SizeInBits AddrExpr2 NumExpr1 NumExpr2 }:
        // Subtract NumExpr1 from the offset of AddrExpr2 and return the
        // resulting address
        Addr <- evalAddrExpr(AddrExpr2, Env, CallStack, GEnv, Mem)
        Offset <- offset(Addr)
        Value <- evalNumExpr(NumExpr1, Env, CallStack, GEnv, Mem)
        Carry <- evalNumExpr(NumExpr2, Env, CallStack, GEnv, Mem)
        NewOffset <- subWCarry(Offset, Value, Carry)
        return addr(fref(Addr), NewOffset, SizeInBits)

    { undefined SizeInBits }:
        return undefAddr(SizeInBits)

    end case
end evalAddrExpr
```

3.3.2 Statements

A program state is defined by a 6-tuple (StmtList, Env, GEnv, Mem, RetAddrList, CallStack). StmtList represents the remaining statements to be executed in the current scope; Env and GEnv are the local and global environments, respectively; Mem is the memory; RetAddrList is a list of addresses to where the current function is to put its results; CallStack is a stack where each element is a 3-tuple—(StmtList, Env, RetAddrList)—that keeps track of previously entered but not yet exited scopes.

Execute one statement in the state (StmtList, Env, GEnv, Mem, RetAddrList, CallStack), and return the next state.

```
execStmt((StmtList, Env, GEnv, Mem, RetAddrList, CallStack))
    // Pop all scopes from the call stack in which the end has been reached
    // (i.e., StmtList = [])
    while StmtList = [] do
        destroyEnv(Env)
        [(StmtList, Env, RetAddrList) | CallStack] <- CallStack
```

```
end while

[Stmt | StmtList] <- StmtList

case Stmt of
{ null }:
    // A null statement does nothing

{ store AddrExprList with ExprList }:
    // (The two lists must have the same length.) Run through the
    // expression list and evaluate the expressions and store them at the
    // corresponding addresses from the address list.
    assert(nrElems(AddrExprList) = nrElems(ExprList))
    while ExprList /= [] do
        [Expr | ExprList] <- ExprList
        Value <- evalExpr(Expr, Env, CallStack, GEnv, Mem)
        [AddrExpr | AddrExprList] <- AddrExprList
        Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
        assert(not isReadOnly(Addr))
        Mem <- store(Mem, Addr, Value)
    end while

{ jump LExpr leaving NrLevels }:
    // Pop as many elements from the call stack as is specified by NrLevels.
    // Then set the statement list (the "PC") to what LExpr is pointing at.
    Label <- evalLabelExpr(LExpr, Env, CallStack, GEnv, Mem)
    assert(not isFunctionLabel(Label))
    while NrLevels > 0 do
        [CSTop | CallStack] <- CallStack
        // Cannot jump out of a function
        assert(not isFunctionScope(CSTop))
        destroyEnv(Env)
        (_, Env, RetAddrList) <- CSTop
        NrLevels <- NrLevels - 1
    end while
    StmtList <- labLookup(Label)

{ switch NExpr TargList }:
    SwValue <- evalNumExpr(NExpr, Env, CallStack, GEnv, Mem)
    assert(onlyOneOccurrenceOfEachCase(TargList))
    FoundMatching <- false
    while TargList /= [] do
        [Targ | TargList] <- TargList

        if isDefaultCase(Targ) then
            { default LExpr } <- Targ
            if not FoundMatching then
                TargetLExpr <- LExpr
            end if
            FoundMatching <- true
        else
            { target Value LExpr } <- Targ
            if Value = SwValue then
```

```
                TargetLExpr <- LExpr
                FoundMatching <- true
            end if
        end if
    end while

    if FoundMatching then
        Label <- evalLabelExpr(TargetLExpr, Env, CallStack,
                                        GEnv, Mem)
        assert(not isFunctionLabel(Label))
        StmtList <- labLookup(Label)
    end if

{ call LExpr ExprList result AddrExprList }:
    // A function call

    // Find function corresponding to label
    Label <- evalLabelExpr(LExpr, Env, CallStack, GEnv, Mem)
    assert(isFunctionLabel(Label))
    { func _ ArgDeclList Scope } <- funLookup(Label)

    assert(nrElems(ArgDeclList) = nrElems(ExprList))
    NewEnv <- _|_
    while ArgDeclList /= [] do
        // Evaulate expression using current Env
        [Expr | ExprList] <- ExprList
        Value <- evalExpr(Expr, Env, CallStack, GEnv, Mem)

        // Create an fref, allocate a frame of size SizeInBits2 that can be
        // referenced by the newly created fref, and store the evaluated
        // expression at offset 0 in the frame
        [{ alloc SizeInBits1 Symbol SizeInBits2 } | ArgDecls] <- ArgDecls
        CounterVal <- readCounter(Symbol)
        incrCounter(Symbol)
        FRef <- fref(frefId(Symbol, CounterVal), SizeInBits1)
        alloc(SizeInBits2, FRef)
        NewEnv <- NewEnv |_| {(Symbol, FRef)}
        Mem <- store(Mem, addr(FRef, 0), Value)
    end while

    // Create a list of addresses to where the called function is to store
    // its return values
    NewRetAddrList <- []
    while AddrExprList /= [] do
        [AddrExpr | AddrExprList] <- AddrExprList
        Addr <- evalAddrExpr(AddrExpr, Env, CallStack, GEnv, Mem)
        NewRetAddrList <- NewRetAddrList ++ [Addr]
    end while

    // Push the current scope, set StmtList to the scope of the called
    // function, and use the new Env and RetAddrList
    CallStack <- [(StmtList, Env, RetAddrList) | CallStack]
    StmtList <- [Scope]
```

```
        Env <- NewEnv
        RetAddrList <- NewRetAddrList

    { return ExprList }:
        // Evaluate returned expressions and store them according to the
        // list of return addresses
        assert(nrElems(ExprList) = nrElems(RetAddrList))
        while ExprList /= [] do
            [Expr | ExprList] <- ExprList
            Value <- evalExpr(Expr, Env, CallStack, GEnv, Mem)
            [Addr | RetAddrList] <- RetAddrList
            Mem <- store(Mem, Addr, Value)
        end while

        // Find the scope in which the function call was made
        // Initialize StackTop with the current visited scope (hm...)
        StackTop <- (StmtList, Env, RetAddrList)
        while not isFunctionScope(StackTop) do
            (_, Env, _) <- StackTop
            destroyEnv(Env)
            [StackTop | CallStack] <- CallStack
        end while

        // Proceed execution of the scope on top of the call stack
        destroyEnv(Env)
        [(StmtList, Env, RetAddrList) | CallStack] <- CallStack

    { free FRefExpr }:
        // Free the dynamically allocated memory pointed to by the evaluated
        // FRefExpr

        FRef <- evalFRefExpr(FRefExpr, Env, CallStack, GEnv, Mem)
        dealloc(FRef)

    { scope Decls Inits { stmts ScopeStmts } }:
        // Enter the scope by pushing the current scope on the call stack and
        // jumping to the statements in the scope

        CallStack <- [(StmtList, Env, RetAddrList) | CallStack]
        StmtList <- ScopeStmts
        (Env, Mem) <- createEnv(Mem, Decls, Inits)

    end case

    return (StmtList, Env, GEnv, Mem, RetAddrList, CallStack)
end execStmt
```

3.3.3 Auxiliary functions

Most of the auxiliary functions that are used in the semantics are left out from this specification, since their names should be self-explanatory. The auxiliary functions that need a deeper explanation are listed in this section.

Create a new environment from the declaration and initialization lists. Returns a pair $(\mathbf{Env}, \mathbf{Mem})$ with a new environment and an updated memory.

```
createEnv(Mem, DeclList, InitList)
    // Allocate frames according to the declarations in DeclList, then introduce
    // mappings from symbols to their frefs
    Env <- _|_
    while DeclList /= [] do
        [{ alloc SizeInBits1 Symbol SizeInBits2 } | DeclList] <- DeclList
        assert(not defined(Env, Symbol))
        CounterVal <- readCounter(Symbol)
        incrCounter(Symbol)
        FRef <- fref(frefId(Symbol, CounterVal), SizeInBits1)
        alloc(SizeInBits2, FRef)
        Env <- Env |_| {(Symbol, FRef)}
    end while

    // Run through all initializations in InitList and initialize the newly
    // allocated frames
    while InitList /= [] do
        [{ init Ref Val VolReadOnly } | InitList] <- InitList
        { ref Symbol Offset } <- Ref
        FRef <- lookup(Symbol, Env, [], _|_)
        if VolReadOnly = volatile then
            setToVolatile(FRef)
        else if VolReadOnly = read_only then
            setToReadOnly(FRef)
        end if

        // Handle the simple case
        if isConst(Val) then
            Mem <- store(Mem, addr(FRef, 0), Val)
            continue
        end if

        // Handle the rest of the cases
        case Val of
        { const_repeat Const Repeats }:
            // Store constant value at address with size interval
            Offset <- 0
            while Repeats > 0 do
                Mem <- store(Mem, addr(FRef, Offset), Const)
                Offset <- Offset + size(Const) / LAU
                Repeats <- Repeats - 1
            end while

        { const_list ConstList }:
            // Store each constant in list. Each store increments address.
            Offset <- 0
            while ConstList /= [] do
                [Const | ConstList] <- ConstList
                Mem <- store(Mem, addr(FRef, Offset), Const)
                Offset <- Offset + size(Const) / LAU
            end while

        { hex_list SizeInBits StringList }:
```

```
        { udec_list SizeInBits StringList }:
        { bin_list SizeInBits StringList }:
            Offset <- 0
            while StringList /= [] do
                [String | StringList] <- StringList
                // Does the value that is represented by String fit in the # of
                // bits that is specified by SizeInBits?
                assert(fits(String, SizeInBits))
                Const <- signed(String, SizeInBits)
                Mem <- store(Mem, addr(FRef, Offset), Const)
                Offset <- Offset + SizeInBits / LAU
            end while

        { dec_list SizeInBits StringList }:
            Offset <- 0
            while StringList /= [] do
                [String | StringList] <- StringList
                // Does the value that is represented by String fit in the # of
                // bits that is specified by SizeInBits?
                assert(fits(String, SizeInBits))
                Const <- unsigned(String, SizeInBits)
                Mem <- store(Mem, addr(FRef, Offset), Const)
                Offset <- Offset + SizeInBits / LAU
            end while

        { float_list SizeInBits1 SizeInBits2 StringList }:
            Offset <- 0
            while StringList /= [] do
                [String | StringList] <- StringList
                // Does the value that is represented by String fit in the # of
                // bits that is specified by SizeInBits1 and SizeInBits2?
                assert(fits(String, SizeInBits1, SizeInBits2))
                Const <- float(String, SizeInBits1, SizeInBits2)
                Mem <- store(Mem, addr(FRef, Offset), Const)
                // The sign bit is not stored
                Offset <- Offset + (SizeInBits1 + SizeInBits2) / LAU
            end while

        { char_string CharString }:
            Mem <- store(Mem, addr(FRef, 0), CharString)

        end case
    end while
    return (Env, Mem)
end createEnv
```

Destroy the environment Env. All frefs whose symbols have a mapping in Env are deallocated.

```
destroyEnv(Env)
    forall Symbol : defined(Env, Symbol) do
        FRef <- Env(Symbol)
        dealloc(FRef)
    end forall
end destroyEnv
```

Search for an fref that has the identifier Symbol. First look in LocalEnv; then run through the call stack from the top to the bottom or down to and including the first scope that has been entered through a function call (since static scoping rules are followed, frefs that were declared before the call are not visible); finally look in the global environment GEnv. Not finding the sought fref is considered an error.

```
lookup(Symbol, LocalEnv, CallStack, GEnv)
    // First look in the local environment
    if defined(LocalEnv, Symbol) then
        return LocalEnv(Symbol)
    end if

    // Then search through the environments on the call stack (note that the
    // call stack is popped in order to iterate through its elements; this does
    // not change the stack that was input as an argument to the function)
    while CallStack /= [] do
        [Top | CallStack] <- CallStack
        (_, Env, _) <- Top
        if defined(Env, Symbol) then
            return Env(Symbol)
        end if
        // Only search until the first scope belonging to a function
        if isFunctionScope(Top) then
            CallStack <- [] // In order to exit the loop
        end if
    end while

    // Finally, the mapping must be defined in the global environment, otherwise
    // something is wrong
    assert(defined(GEnv, Symbol))
    return GEnv(Symbol)
end lookup
```

3.3.4 Semantics of a whole ALF program

The following describes the semantics of an entire ALF program as the resulting state of the memory after the entire program has been executed.

```
execProg(State)
    (StmtList, Env, GEnv, Mem, RetAddrList, CallStack) <- State

    // Run as long as there are some statements left in the current scope to
    // execute or there are more scopes to execute
    while StmtList /= [] or CallStack /= [] do
        (StmtList, Env, GEnv, Mem, RetAddrList, CallStack) <-
            execStmt((StmtList, Env, GEnv, Mem, RetAddrList, CallStack))
    end while
    return Mem
end execProg
```