Translation of the StrictC Dialect

Michael Norrish

20 September 2015

Contents

1 Introduction

This report describes the translation program that imports C source into a running Isabelle/HOL process, making a series of Isabelle definitions in the process, as well as discharging a number of (relatively minor) proof obligations that arise along the way.

The source code for the translator is found in the directory `c-parser`. Source files, *e.g.*, `file.ML`, are found in this directory unless otherwise noted.

The translation expects its input to be a well-formed C source file. Such a source file must additionally satisfy a number of other constraints, giving rise to a subset of C that is here called "StrictC". Files in StrictC may also include

special annotations intended only for consumption by Isabelle (and the human code-verifier).

In fact, there are two important StrictC programs: the translation program, and the analysis program. The analysis program is entirely independent of Isabelle, and can be used to check that a source file conforms to the StrictC subset. It also implements a number of analyses that can be performed on source code. For example, it can output the input file's call-graph, and can list the globals that are read or modified in each function. The additional source-files supporting the analysis program are found in

```
c-parser/standalone-parser
```

The rest of this report describes both the functionality provided by these programs (focussing on the translator), how this functionality is implemented, and *where* it is implemented. The aim is to give a picture of the systems' design in a way that should make future modification of the code possible.

1.1 StrictC Subset Summary

This is a brief list summarising the simplest restrictions imposed by the StrictC subset.

- No `goto` statements.
- No fall-through cases in `switch` statements. Cases can be terminated with `continue` and `return`, as well as `break`.
- Labels for `switch` statement cases must all appear at the syntactic level immediately below the block statement that must appear below the `switch` statement.
- No unions. (These are handled by a separate tool; see Cock [1].)
- No `struct`, `enum` or `typedef` declarations anywhere except at the top, global, level.

2 Abstract Syntax

The core data types in the translator represent the input program. These abstract syntax values are the product of parsing the concrete syntax (see Appendix A for the grammar used), and is the subsequent input to all further analyses and translation. The abstract syntax declarations are given in `Absyn.ML`. For example, the definition of the syntax type corresponding to C statements is given in Figure 1. There are also definitions for C types, expressions, and declarations in `Absyn.ML`.

The type `statement` is actually a `statement_node` wrapped inside a "region" (see `Region.ML` and Section 2.1 below), which provides information about

```
datatype statement_node =
      Assign of expr * expr
    | AssignFnCall of expr option * expr * expr list
    | EmbFnCall of expr * expr * expr list
    | Block of block_item list
    | While of expr * string wrap option * statement
    | Trap of trappable * statement
    | Return of expr option
    | ReturnFnCall of expr * expr list
    | Break
    | Continue
    | IfStmt of expr * statement * statement
    | Switch of expr * (expr option list * block_item list) list
    | EmptyStmt
    | Auxupd of string
    | Spec of ((string * string) * statement list * string)
    | AsmStmt of {volatilep : bool, asmblock : asmblock}
and statement = Stmt of statement_node Region.Wrap.t
and block_item =
    BI_Stmt of statement
  | BI_Decl of declaration wrap
```

Figure 1: The Abstract Syntax Data Type for C Statements

where the original concrete syntax originated. This is used for providing error messages.

All strings in the statement declaration correspond to Isabelle terms (*e.g.*, the loop invariant in the `While` case). These will be parsed as such later in the translation process, but are just uninterpreted strings when the C parser finishes.

The statement type illuminates StrictC's most radical departure from ISO C: the movement of assignment and function calls from the category of expression to statement.[1] This change syntactically enforces the requirement that expressions must be side-effect free. Both forms must occur at the top-level. Function calls can return their results into l-values, have the return value ignored, or have the return value itself `return`-ed. The first option corresponds to having the `expr option` argument be `SOME e` in the `AssignFnCall` constructor, the second would have that parameter be `NONE`, and the last is handled by the `ReturnFnCall` constructor.

Syntactically, these options correspond to writing

```
var = f(x,y);
```

or

```
f(x,y);
```

or

```
return f(x,y);
```

C99 Block Items Conforming to the C99 grammar, the input language allows declarations at any point inside a block, not just in a sequence at the head of the block. In other words, the following

```
{
  x = f(z);
  int y = x + 1;
  while (x < y) { ... }
}
```

used to be illegal, but is now legal C. This means that a block has to take a list of `block_item` values as an argument, where a `block_item` is either a statement or a declaration.

Syntactic Sugar for Loops The abstract syntax has just one form of loop, the `While` constructor. The optional string argument to `While` is used to represent any user-supplied invariant. Together with the `Trap` constructor, this is used to implement all three forms of C loop. The translation follows the model from Norrish's PhD [3, p60]. It also supports `for`-loops with declarations in the first position. This latter, for example,

[1] As is not documented but should be, function calls may seem as if they are permitted inside expressions, but are not really.

```
for (int i = 3; i < 10; i++) ...
```

is a feature of C99.

Breaking from C, the grammar has the third component of the `for` loop form be a restricted form of statement, rather than an expression. The parser syntax only allows comma-separated increments (*i.e.*, `++`), decrements and assignments.

The Isabelle translation eventually compiles all loops to one underlying the loop primitive in the VCG environment called `While`. This form does not handle exceptional control-flow forms like `break` and `continue`. These are instead handled by the `Trap` constructor, mapping to the VCG language's `TRY-CATCH` form.

2.1 Regions

The `Region` module implements a method for annotating arbitrary data types with location information. This module has been taken from the `MLton` compiler project (which has a BSD-style open source licence). It is used a great deal in the system, and its use could probably be extended still further. Region information is used to produce good error messages.

The basic type is that of the `region` which is essentially a pair of "source positions" (which are in turn implemented in `SourcePos.ML`). One useful source position is `SourcePos.bogus`, corresponding to "nowhere" (perhaps because some syntax has been conjured out of nowhere and doesn't really exist in a file).

Regions are then used to implement the concept of a "wrap" (SML type `Region.Wrap.t`), a polymorphic data type. The file `Absyn.ML` declares the following abbreviation:

```
type 'a wrap = 'a Region.Wrap.t
```

An `'a wrap` (read "alpha wrap") is an `'a` value coupled with a region. The important functions for manipulating wraps are

```
val wrap    : 'a * SourcePos.t * SourcePos.t -> 'a wrap
val bogwrap : 'a -> 'a wrap
val left    : 'a wrap -> SourcePos.t
val right   : 'a wrap -> SourcePos.t
val node    : 'a wrap -> 'a
val apnode  : ('a -> 'b) -> 'a wrap -> 'b wrap
```

For example, the grammar code in `StrictC.grm` manipulates a number of values of type `string wrap`. If an error relating to this value arises, both the string and its position can be reported to the user.

Things become more complicated when the type to be wrapped is recursive. The standard idiom in the project is illustrated with the definition of the `statement` data type (shown in Figure 1). The constructors for values of the type are actually given in an auxiliary type (`statement_node`), but the recursive constructors take arguments of type `statement`. The type `statement` is then a

type that is mutually recursive with `statement_node`, and which has just one constructor, `Stmt`.[2]

Because a `statement` is not a wrap, the project cannot directly call functions like `node` on `statement` values. Instead, helper functions are declared:

```
val sleft  : statement -> SourcePos.t
val sright : statement -> SourcePos.t
val snode  : statement -> statement_node
val swrap  : SourcePos.t * SourcePos.t * statement_node ->
             statement
```

When code wishes to pattern-match against the multiple possible forms a statement `s` may have, the idiom is to write

```
case snode s of
  EmptyStmt => ...
| While(g,inv,body) => ...
| IfStmt(g,ts,es) => ...
| ...
```

The strength of this idiom is that one *always* manipulates values of type `statement`. In particular, if the case analysis above is to make recursive calls of its analysis on sub-statements such as `body`, `ts` and `es`, these values are of the correct type for this to be done immediately.

The `expr` (expression) type (home to constructors such as `Deref`, `Var` and `TypeCast`) is set up in the same style, giving rise to functions `eleft`, `eright`, `enode`, `ewrap` and `ebogwrap`.

The type of C types The type of C types is `'a ctype`, with constructors such as `Void` and `Ptr`. Though recursive, it doesn't use the wrap idiom. On the other hand, this type is polymorphic. The `'a` parameter is instantiated with an SML type that corresponds to the forms that give the size of arrays when they are declared. This type parameter is instantiated with `expr` when the input file is first parsed. In this way, a declaration like

```
unsigned char array[EnumConst1 * sizeof(int*)];
```

can be handled. Thus, the `VarDecl` constructor of the `declaration` type

```
val VarDecl :
      expr ctype * string wrap * bool * initializer option ->
      declaration
```

takes an `expr ctype` as its first parameter. Within subsequent phases of the analysis and translation, it is much more convenient to work with values of type

[2]It would be nice if one could just declare `statement` to be an abbreviation of `statement_node wrap`, but SML doesn't permit this. It must be a `datatype` itself, and thus must have at least one constructor.

`int ctype`, where the (constant) expression has been evaluated. For example, the `get_rettype` function from `program_analysis.ML`, takes a `csenv` value (see Section 3 below) and the name of a function, and returns the return type of the function. The value returned is an `int ctype`. The conversion of an `expr ctype` into an `int ctype` is done by the function `Absyn.constify_abtype`.

3 The Symbol Table

All of the work done in analysis and translation of StrictC revolves around the information stored in two important data structures implemented in the module `program-analysis.ML`. The first of these is the `var_info` type. This stores information about individual variables. The second type, `csenv` ("C state environment"), stores information about the program as a whole, including its collection of variables, but also recording details such as where variables are read and modified, and the program's call-graph structure.

The `var_info` type stores information about declared identifiers living in the name-space that encompasses normal objects, functions and enumeration constants. In addition to the type of the variable (*e.g.*, `int`, `char *` *etc.*), the `var_info` also includes information about where the variable was declared in terms of program locations, and also in terms of scope (it might be global, or declared local to a particular function).

The `csenv` type accumulates its information about the program by performing traversals of the abstract syntax tree. **The StrictC translator makes no effort to be a one-pass compiler**, but the number of traversals is no greater than three, and will probably be reduced in future versions of the implementation. These traversals are performed after the parser has constructed all of the tree. There are also places in this analysis where **the translator assumes that it has seen the whole program**. In particular, the translator cannot be used to translate *translation units* that are to be separately compiled. It must be presented with a concatenation of the complete sources.

The bulk of the API for manipulating values of type `csenv` is concerned with pulling information out of the symbol table. For example, it is possible to calculate the type of a C expression with the function

```
val cse_typing : csenv -> expr -> int ctype
```

In addition, `program-analysis.ML` contains the one entry-point for taking a sequence of external declarations (once parsed) and creating a `csenv` value:

```
val process_decls :
  Absyn.ext_decl list ->
  ((Absyn.ext_decl list * Absyn.statement list) * csenv)
```

The return type includes a modified version of the syntax that was provided as input, a list of the initialising assignments for the global variables, and the `csenv` value.

3.1 Functional Record Updates in SML

The code in `program-analysis.ML` uses a powerful, but cryptic SML idiom that makes it easy to define SML records along with functions for updating their fields. Done naïvely, writing code to do this represents a quadratic amount of work for the programmer. Put another way, the naïve approach requires $O(n)$ much typing whenever a field is added to or removed from a record definition of n fields.

The cryptic technique is fully described at

`http://mlton.org/FunctionalRecordUpdate`

and allows the addition or deletion of a field to be done with $O(1)$ much typing. Supporting code is in `FunctionalRecordUpdate.ML`.

The cryptic code is isolated within `program-analysis.ML`, where it is used to define update functions that are subsequently used exclusively. In general, when one of the two types has a field `fld` of type τ, then there will typically be a function {`cse`|`vi`}`_fupd_fld` defined, with type

$$(\tau \to \tau) \to \mathsf{rcd} \to \mathsf{rcd}$$

where rcd is either `var_info` or `csenv`. Such functions can be used to update the fields of a record: the user provides a function that is given the old value of the field, and which returns the new value.

4 Creation of the Hoare Environment State

The major oddity about Norbert Schirmer's Hoare environment, into which we are translating our programs, is that all local variables, including function parameters, have to be part of the "global state". This state must be declared before any functions can be translated, because a function becomes an Isabelle definition (conceptually at least) that operates over that state space.

Slightly simplifying, the state space is an Isabelle type that is a record with fields for every local and global variable. Each field has a type (Isabelle/HOL is a typed logic after all), which means that all local variables of the same name in the same StrictC translation unit must have the same type. This is rather an arbitrary requirement, but easy both to enforce and to comply with. Thankfully, signed and unsigned variants of the same underlying type (such as `signed short` and `unsigned short`) are given the same Isabelle/HOL type, so there is a little leeway. Nonetheless, if `i` is of type `int` in one function, it can not be of type `char` in another.[3]

The arrangement of global and local variables is actually slightly complicated. In essence, the state-space is set up to look like:

[3] If this is attempted, the system will "munge" one of the variables so that it has a different name when translated into Isabelle. The "munged" name is stored in the variable's `var_info` record.

```
statespace = record
   globals :: global_var_type
   local_var1 :: lvar1_type
   local_var2 :: lvar2_type
   ...
   local_varn :: lvarn_type
```

where the field `globals` is of a custom record type `global_var_type` that in turn contains all of the global variables. In addition to the user program's own globals, the translation process adds two extra global variables of its own. These variables are used for handling "exceptional exits" (such as those caused by the `break`, `continue` and `return` statements), and for modelling the global heap. The exact names of these variables is not important, here we will refer to them as `global_exn_var`, and `global_heap`.

These two special variables are part of `global_var_type` and so must have Isabelle types themselves. The type of `global_exn_var` is the enumerated type `c_exntype`, defined in the Isabelle theory `CProof` to have three possible values, `Break`, `Return` and `Continue`. The type of the global heap `global_heap` field is a product of underlying heap contents (a map of type `addr` $\rightarrow$ `word8`) and a special-purpose data type to store type information about the heap memory (see Harvey Tuch's PhD thesis [4] for more on this).

4.1 Representing Values in Memory

There is one important requirement that must be met by all object types that occur in C programs: they must be representable in memory. Alternatively, it must be possible to encode values of the types in a program as sequences of bytes, and to then decode those same bytes back into the original values. One approach to modelling this fundamental requirement might be to have Isabelle functions that manipulated only lists of bytes. Working at the level of this untyped, and very concrete, view of the program state would be an extremely poor state of affairs (the C programmer would have a more abstract view of the program than the verifier).

Our approach is to use Isabelle type-classes to encode the fact that an Isabelle type can be represented in a consistent amount of "C memory". When an Isabelle type `'a` is in the class `mem_type`[4], it supports functions

```
to_bytes   :: "'a::mem_type => byte list"
from_bytes :: "byte list => 'a::mem_type"
```

as well as a number of other supporting functions that record details like the fields that occur in compound `struct` types, and the (constant) length of the byte-lists that encode the values in the type.

All of the atomic types manipulated by our programs are fixed-width words (*e.g.*, 32 bit words for integers). It is easy to demonstrate that these types are

[4] See `umm_heap/CTypes.thy`.

indeed in the `mem_type` class. In particular, we do *not* pretend that programs manipulate infinite precision integers, and then worry about whether or not these integers can be pushed into and pulled out of memory. All of the arithmetic performed by the programs we verify is done at fixed widths, respecting the underlying machine's operations. Additionally, using the techniques from Section 5.1, we trap the undefined behaviour caused by overflow on signed values.

4.2 Pointers

Pointers are always represented as words of a particular size (regardless of type being pointed to). This is not required by the C standard, which only requires pointers to `void` be capable of storing all other non-function pointer values, and that all function pointer values be inter-convertible. Again, our decision to specialise on particular, and reasonable, target architectures makes life simpler.

Pointers retain type information by using "phantom" type variables. The Isabelle declaration is

```
datatype 'a ptr = Ptr addr
```

Then, if the C program under analysis calls for a variable of type `char *`, the Isabelle environment will include a variable of Isabelle type `byte ptr`. In this way our pointers are typed, even if their underlying encoding makes it trivial to view a pointer to one type as a pointer to another type.

Because the underlying representation is always the same, all pointer types are proved to be in the class `mem_type` once and for all (in theory `CTypes`). Pointers to `void` are represented as values in the Isabelle type `unit Ptr`, where `unit` is the standard singleton type. The `unit` type is not shown to be in the class `mem_type`.

4.3 Arrays

C arrays are lists of values of fixed length. A faithful representation of this type requires a novel Isabelle type. We build on Anthony Fox's implementation of John Harrison's "finite Cartesian products" idea [2]. Syntactic trickery within Isabelle allows us to write types like

```
nat[10]
```

which is an array of 10 natural numbers. There are operators for updating and indexing into arrays. Note that the type `10` that appears above is an Isabelle *type*, not a term.

If type τ is a `mem_type`, then an array of τ values will also be a `mem_type`, as long as the size of the array is not so large that the array would not fit into memory. This condition is discharged as the StrictC program is translated. For technical reasons due to the implementation of type classes, the restriction on sizes is implemented with two restrictions that must both be met:

```
struct listnode {
   int node_data;
   struct listnode *next;
};
```

```
struct node2;
struct node1 {
   struct node2 *data;
   struct node1 *next, *prev;
   int someflag;
};
struct node2 {
   struct node1 *owner;
   char stringdata[100];
};
```

Figure 2: Examples of Recursive `struct` Declarations Accepted in StrictC

- arrays must not have more than 2^{13} (8192) elements; and
- array elements must not have sizes greater than 2^{19} bytes (512 kB)

This is not ideal. One would prefer to be able to multiply the size of the element type by the number of the elements, but the type system does not permit this (for good technical reasons).

4.4 C `struct` Types

From the point of view of the translation into Isabelle, `struct` types do not pose any great conceptual difficulties. A `struct` type is clearly very similar to an Isabelle record, which in turn is conceptually the same as a tuple. The first complication that arises is that Isabelle tuples can not be recursive, whereas C `struct` types are often recursive (as when implementing linked structures in the heap).

This required the implementation of an alternative record definition package, allowing (possibly multiple) recursive types. This then allows Isabelle types to be defined that correspond to C declarations such as those shown in Figure 2.

Confirming that a `struct` type really is representable in memory, requires the definition of functions for converting Isabelle records into lists of bytes and *vice versa*. The size of the converted value must also be checked to be no bigger than the size of memory. Both of these actions require knowledge of how the fields of the `struct` are laid out in memory, which is in turn a function of the padding that can be inserted between the fields. Such calculations are architecture dependent.

5 Translating Expressions

Expression translation is implemented in `expression_translation.ML`.

Fundamental Concepts StrictC expressions are essentially a subset of C expressions, and are fairly easy to translate to corresponding Isabelle expressions

```
datatype expr_info =
  EI of {lval   : (term -> term -> term) option,
         addr   : (term -> term) option,
         rval   : term -> term,
         cty    : int ctype,
         ibool  : bool,
         lguard : (term -> term * term) list,
         guard  : (term -> term * term) list,
         left   : SourcePos.t,
         right  : SourcePos.t }
```

Figure 3: The `expr_info` type, into which C expressions are translated internally.

that manipulate Isabelle-encoded values. There are two fundamental concepts to grasp of expression translation. First, when being evaluated ("read to determine a value") a C expression of type τ becomes an Isabelle expression of type `statespace` $\rightarrow [\![\tau]\!]$, where $[\![\tau]\!]$ is the translated, Isabelle version of C type τ.

This must be done to make sense of expressions that read memory: an expression such as `s.arrayfld[3]` only has a specific value in the context of a specific state of memory. This use of a "lifted" function space to represent the expression is a standard technique in denotational semantics. In the example given, the value of the expression is a function that looks at the statespace to determine what data is stored at variable `s`. As the statespace evolves, the value returned from this function changes, but the function's value is the same.

Expressions do not just determine values however, they can also denote an *l-value*, something denoting a "place in memory" that is to be updated. This is done when an address is taken, or when an assignment is to be performed. In a simple language, l-values might only be variable names, but C allows for complicated expressions on both sides of an assignment. The example above (`s.arrayfld[3]`) might just as well be assigned to as read. So, the l-value of an expression e that has type τ will be an Isabelle value of type `statespace` $\rightarrow$ $\tau \rightarrow$ `statespace`, a function that takes a new value and a statespace to change, and returns the updated statespace.

Not all expressions are l-values (the expression `3` is not, for example), so the translation of any one expression can return one or two different values, an "r-value" as well as an optional l-value.

In addition, because of the way the translation does not put local variables into memory, the translation provides another separate optional value, that of the expression's address. If everything did live in memory, all l-values would have addresses, and one could dispense with the separate calculation of l-values. Thus the first three lines of Figure 3.

The SML types given to the `lval`, `addr` and `rval` fields in the declaration of `expr_info` are themselves function spaces at the SML level. These function

spaces manipulate values of SML type `term`. The way in which typing at the C level is reflected at the Isabelle level is mainly hidden at the SML level, where the programmer just manipulates terms (the Isabelle types are internal to those values).

However, the function spaces *can* be made visible at the SML level. This is done mainly to reduce the number of β-redexes that would otherwise be created in the resulting term. For example, translating the C expression `x + 3` will first create r-values

$$\begin{aligned} [\![\mathtt{x}]\!] &= (\lambda\sigma.\, \mathtt{x}(\sigma)) \\ [\![\mathtt{3}]\!] &= (\lambda\sigma.\, 3) \end{aligned}$$

where the application of the `x` function to a statespace σ pulls out the value of variable `x` in σ.

When translating an expression $e_1 + e_2$, one naturally creates the value

$$\lambda\sigma.\, [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$$

If this was done within Isabelle `term` values, the result would be a `term` full of expressions of the form $(\lambda v.\ M)N$. By lifting the Isabelle λ to the SML level, only one abstraction need to be created, at the very top-level in the translation. Thus, the translation of the addition becomes the SML expression

```
(fn s => mk_plus (rval_of e1 s) (rval_of e2 s))
```

where `rval_of` returns the `rval` field of an `expr_info` and the β-reduction happens within SML.

The fourth line of the record in `expr_info` values stores the type of the expression, something that informs the translation of many different C expressions. For example, additions are not as simple as just presented because of the possibility that one of the arguments might be a pointer. The `left` and `right` fields of the `expr_info` record store the source-code position of the original expression. The two guard fields are explained below in Section 5.1.

The `ibool` field of the `expr_info` records whether or not the term being generated in the r-value is of Isabelle's boolean type. This is done so that translation can avoid some conversions between Isabelle words and booleans. For example, if the expression being translated is `x < 6 && y > 10`, the resulting Isabelle term will include a use of the two comparison operators on words. Strictly, one should then turn the boolean results of these operators into either a one or zero. But, as these results are then combined with boolean conjunction, the values will immediately be converted back into booleans.

Of course, in an expression such as `x && x->fld > 6`, the first operand to the conjunction does not have Isabelle boolean type, and will be converted to a boolean value by comparing it with the null pointer. (In fact, this particular example causes a more complicated translation effect to occur; see Section 5.1 below on undefined behaviour.) Dually, if the code puts a boolean value into memory, the translation has to make sure that the Isabelle term has the appropriate word type once more. There are two functions used here:

```
mk_isabool : expr_info -> expr_info
strip_kb   : expr_info -> expr_info
```

The function `mk_isabool` produces an `expr_info` value that does have Isabelle boolean type (it will be the identity function on an r-value that is already known to be boolean). The function `strip_kb` reverses this, turning an Isabelle boolean into an Isabelle word if necessary.

5.1 Undefined Behaviour

There are three classes of under-specification in the C standard. Those classed as *implementation defined* are behaviours or values that are supposed to be fixed and documented by particular implementations. For example, the number of bits in a byte (a number that must be at least eight), is a fixed value that implementations are allowed to choose themselves. Because StrictC targets a particular architecture, most implementation-defined aspects of C can be "baked into" the translation. (The design attempts to have the translation sources be easy to modify to account for different architectures; see for example the `IntInfo` structure in `termstypes.ML`.)

The second class of under-specification comprises *unspecified* behaviours. The most important of these is the order of evaluation of arguments to operators and function calls. Our translation completely bypasses this problem by removing side effects from expressions, meaning that expressions can be evaluated in any order because they will all derive values based on the same underlying state.

The third class of under-specification is *undefined behaviour*. In essence, undefined behaviours are runtime errors (such as dereferencing a null pointer). They are undefined because the standard does not require the implementation to catch them, or to realise that they have occurred. Rather, if an undefined behaviour occurs, the user can no longer rely on their program to do anything sensible. In effect, implementations are given licence to blunder on however they like when an undefined behaviour occurs.

Thus, it is critical that the C code we verify never exhibits any undefined behaviours. We do this with a feature of the Hoare environment called the *guard*. A guard is a boolean condition g attached to a statement s, with the combination written $g \to s$. If the guard g is true when the combined statement is to be executed, then s is allowed to execute. If it is false, the underlying semantics defines the result to be a "fault". When a program faults, nothing more can happen, so it has effectively aborted. Verifying a program with guarded statements thus requires proofs that the guards are always true.

Most guards arise in expressions, and the translation process accumulates these in one top-level guard at the statement level. For example, in the statement

```
x = *p >> i;
```

there will be four guards attached to the statement that will need to be discharged: that `p` is not null, that `*p` is not negative, that `i` is not negative, and that `i` is less than 32.

One elegant feature of guards in the Hoare environment is that they can be selectively disabled for the purposes of verification. For example, the C standard's requirement that right shift operations might not be performed on negative numbers (that it results in undefined behaviour) is too strict, given a particular C compiler and target architecture. In this situation, it is possible to prove Hoare-triples where that particular guard is not used, so that the semantics of $g \rightarrow s$ is simply that of s.

A Concrete Syntax: Parsing and Lexing

The grammar for StrictC is given in the file `StrictC.grm`, which is given as input to the standard tool `mlyacc`. The format of an `mlyacc` file looks quite similar to the format accepted by `yacc`. A typical grammar rule is

```
init_declarator_list
: init_declarator
                ([init_declarator])
| init_declarator YCOMMA init_declarator_list
                (init_declarator :: init_declarator_list)
```

where a non-terminal appears before a colon, and multiple possible right-hand sides are separated by the pipe or vertical bar character. The code for a production appears in parentheses after each right-hand-side. The code's convention is to have token names (*e.g.*, `YCOMMA` above) be upper-case.

Apart from the changes that turn assignment expressions into statements, the grammar for StrictC attempts to be as close as possible to the grammar of the C standard. In general, the names for non-terminals in `StrictC.grm` are taken from the standard, so it should be fairly clear how the standard's grammar has been mapped into `StrictC.grm`.

A.1 Lexing and `typedef` Names

The standard problem in lexing and parsing C is that the grammar is ambiguous: the non-terminal *typedef-name* is defined to simply be the same as an *identifier*. When the parser encounters

```
x * f(y);
```

it can't know if this is meant to be a multiplication of variable `x` by a function call expression, or whether it is the (prototype) declaration of a function called `f` taking an argument of type `y` and returning a value of type `x`.

To resolve this, the lexer must be able to classify identifier tokens as normal identifiers or *typedef-names*. The StrictC translator's approach to this problem is not typical, because of the strange way in which `mlyacc` combines handling

of side effects with its error correction. Normally, one would have some sort of updatable symbol table that the parser would write to when it encountered a `typedef` declaration. The lexer would read from the same table as it encountered identifiers, allowing appropriate categorisation.

The approach taken in the StrictC translator is to have the lexer do all of the work, without reference to the parser (see `StrictC.lex`). This *is* possible, but is also convoluted, and involves many updatable variables that are internal to the lexer. The basic idea is that when the lexer sees a `typedef` token, it switches to the `TDEF` state. When an identifier is seen in this state, the identifier is added to the list of *typedef-names*, and lexing can continue. The complications arrive in declarations like

```
typedef struct s { int fld1; char fld2; } s_t;
```

where the identifiers `s`, `fld1` and `fld2` have to be ignored, and `s_t` taken as the new *typedef-name*. This requires the lexer to handle the matching brace characters, and partly motivates the requirement that `typedef` declarations all occur at the top-level (and not be nested).

A.2 GCC `__attribute__` Declarations

The parser handles, but mostly ignores, various `gcc`-specific extensions, such as `__attribute__`. These are tricky to parse (something admitted by the relevant `gcc` documentation): users are given almost unlimited liberty to put their `__attribute__` strings anywhere within a declaration.

For example, these three

```
int f(int) __attribute__((__const__));
__attribute__((__const__)) int f(int);
int __attribute__((__const__)) f(int);
```

are all supposed to parse successfully (and have the same meaning). Making this work is rather involved. Most attributes are ignored, but the standalone analysis tool does check that `const` and `pure` attributes are reasonable, given what it knows of how functions may modify and read the global state.

References

[1] David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *Proc, 5th VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, August 2008.

[2] John Harrison. A HOL theory of Euclidean space. In Joe Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2005.

[3] Michael Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as Technical Report 453, available from `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf`.

[4] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, Aug 2008.