Sparse: a look under the hood

June 8, 2016
This article was contributed by Neil Brown

"Sparse" is a C language "semantic parser" originally written by Linus Torvalds to support his work on the Linux kernel. It was designed, according to the README file, to be "small - and simple" and particularly to be "easy to use". Reasons to use a simple C parser could include data mining (to summarize particular features of some code, for example), analysis (possibly to look for troublesome patterns), or visualization (to make it easier to understand or navigate around a large code set). In support of this reuse, sparse is licensed under the permissive MIT License and is structured as a library that other tools can easily incorporate. This library is accompanied by a number of tools that demonstrate some of those reuse possibilities.

Unfortunately though, sparse comes with little documentation to help a potential user get started. In the hope of correcting this omission, the following is an attempt to make the internals for sparse more approachable and to highlight some of the various uses that can be made of it.

Background patterns

Before getting into the details of the interfaces, some observations on overall style will be helpful. The first point to note is that sparse makes free use of global variables. Various aspects of the current state of the parser, and various configuration options set from the command line, are stored in global variables. Consequently, many functions have side effects that are not at all obvious at first glance, so caution and thorough research is advisable when exploring the code. Beyond this general pattern, there are two particular details of sparse worth exploring: memory allocation and list manipulation.

For memory allocation, sparse does not use the familiar `malloc()` and `free()` interfaces from the C library but, instead, provides a dedicated allocator that uses `mmap()` to allocate large blocks of memory which are then subdivided. In an approach somewhat reminiscent of the "slab allocator" used in the Linux kernel, the sparse allocator allocates multiple "blobs" that are each used for a distinct type of object such as tokens, identifiers, expressions, etc. The allocator is optimized for a usage pattern characterized by lots of allocations with few or no deallocations happening until a time comes where *all* objects of a particular type are released. Freeing individual objects is supported for fixed-sized allocations only, in which case the freed space is simply placed on a free list to satisfy a subsequent allocation request.

The wholesale freeing of all objects of a given type typically happens after a particular file has been completely processed, thus allowing a number of files to be processed sequentially without needing to store all of them in memory at once. When multiple files are processed, there may be some preamble that should apply to each file. The `-D` command-line option, which provides an initial definition of a macro, is a common example, but there are others, such as `-include`, which identifies a file to be included before the main file. In order to preserve the results of processing this preamble when memory is freed after processing the first file, a `protect_foo()` interface is provided for each allocator. This interface ensures every "foo" allocated so far will never be freed. This is used after the prefix has been parsed to preserved those results indefinitely.

The `foo` in `protect_foo()` above is any of the different defined allocators, of which there are 20. Sparse uses the C pre-processor to effect a mechanism similar to C++ templates so that allocators can be defined that are type-safe, returning or consuming a particular type rather than just a `void` pointer as `malloc()` and `free()` use. This leads to one of my personal least favorite coding styles, where the definition of a

function, being constructed inside a macro, cannot be found by `git grep` or `etags`. A search for a string like "`__alloc_statement`" finds the single use of this function, but does not report its definition. We will get back to this problem later.

The generic lists used in sparse are quite different from the "list_head" based lists used in the Linux kernel. They consist of a simple linked list of arrays of pointers. This two-level structure (list of arrays) makes iteration over the list a little more complex, but means that generic insert, delete, and concatenate operations can be performed efficiently, and that direct indexing is straightforward after a linearization step.

Parsing phases

The `README` file describes the parsing phases as:

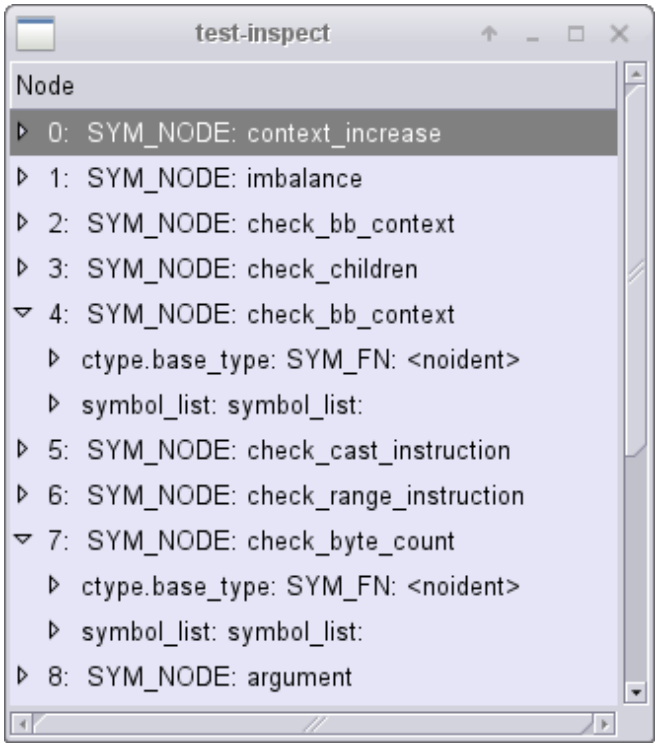
- full-file tokenization
- pre-processing (which can cause another tokenization phase of another file)
- semantic parsing
- lazy type evaluation
- inline function expansion and tree simplification

which is reasonably accurate, but is not quite what you see when you look in the code. From the perspective of a program making use of the sparse library there are three main phases and two high-level data types that need to be understood.

The first phase is embodied in the `sparse_initialize()` and `sparse()` functions. The former is given the command-line arguments (`argc`, `argv`) passed to the program and an empty list. The command-line arguments are processed to emulate `gcc` or a similar compiler, so macro definitions (`-D`), warning levels (`-W`), machine types (`-m`), and preliminary include files (`-include`) are all handled, among others. All target file names are added to the passed-in list, all flags that modify global state are reflected in the relevant global variables, and all preliminary code, such as `-D` and `-include`, is parsed to produce a list of symbols which is returned. The parsing also modifies global variables such as `hash_table[]`, which stores all identifiers.

The list of symbols represents all the top-level definitions of functions and variables. Declaration of types, typedefs, and references to

external functions do not appear in this list: the relevant details will be found within the substructure of the symbols where these declarations are used. Sparse comes with a tool called `test_inspect` that allows the symbol list parsed from a given file to be displayed and some of the substructure of each symbol to be inspected. This is useful for getting a feel for what sparse is producing.



Once `sparse_initialize()` has been called and the file list is no longer empty, `sparse()` can be called in turn on each file in that list. This will parse each file in the context extracted from the arguments and return a separate `symbol_list` for each file. The `symbol_list` is the result of nearly all of the parsing phases listed earlier.

`tokenize_stream()` is first used to convert a `stream` (an abstraction over either a file or a text buffer) into a linked list of `token`s (not using the generic list framework, just a simple linked list). Each token includes a `position` so that the results of parsing, such as warnings, can be accurately linked back to the original code.

The token list, once completely extracted, is passed to `preprocess()`, which performs the various substitutions expected of a C preprocessor. While this is a distinct, well defined phase, the call to `preprocess()` is hidden inside the `sparse_tokenstream()` function which then repeatedly calls `external_declaration()` on the stream of preprocessed tokens to add declarations to a global `hash_table` and build the list of external symbols.

At this stage, the detail within each symbol is just an abstract syntax tree representation of the parsed code. There are statements, expressions, argument lists, and all the details that can be extracted from a purely syntactic analysis. The only analysis that has been performed beyond local syntax is the connection of the use of each symbol to its declaration. This work is necessary because, as discussed in the Wikipedia article on "The lexer hack", correct

syntactic analysis of C requires that symbols declared by typedef be distinguished from other symbols.

There is one more parsing step applied to the symbol list before it is returned by the `parse()` entry point: `evaluate_symbol()` is called. This combines the "lazy type evaluation" and "inline function expansion" phases mentioned in the `README`. It resolves details of the type of each symbol such as storage size and alignment and then checks all initializers and code for type compatibility. This determines the type for all expressions, or reports errors and warnings when unacceptable or undesirable constructs are found. Exactly which warnings are generated here and which are left until later seems a little *ad hoc*. For example, sparse produces a warning if a simple assignment is found in the condition of an `if` statement:

```
if (variable = value)
   statement;
```

While this could be detected during `sparse_tokenstream()`, it is actually handled in `evaluate_symbol()`. This is a common pattern: functionality is often to be found somewhere convenient rather than somewhere meaningful. This doesn't affect the functionality of the code, but can detract from its transparency.

Tree simplification

Based on the parsing phases listed in the `README`, all that is left after the call to `sparse()` is "tree simplification". This simplification happens in two stages that must, if wanted, be called by the main program after the call to `sparse()`.

`expand_symbol()` can be called on each symbol in the list and primarily performs constant folding. For example if it finds an expression that adds "3" to "4" it will replace it with the constant "7". One detail that the parser tracks is whether a given symbol has ever been assigned to or otherwise had its value changed. If a symbol has an initializer but has never been changed, then `expand_symbol()` will use the initialized value where ever the symbol is found, thus achieving a higher level of code simplification.

`expand_symbol()` does a little bit of dead-code elimination when the constant propagation determines that `&&`, `||`, or `?:` can only have one possible outcome. For example, if sparse encounters code like:

```
3 - 3 && foo()
```

the call to `foo()` will be removed, since it will never be executed. Dead code caused by a construct that resolves to "`if (0)`" is not eliminated at this stage as a jump into the body of the `if` could keep some of the code alive.

Finally, `expand_symbol()` performs a little bit of optimization. If a conditional expression (one using the "`? :`" operator) is found to have no side effects, and its computation is not expensive, then the type of the expression is changed from `EXPR_CONDITIONAL` to `EXPR_SELECT`, implying that it can be implemented without using any jump instructions, since a `cmov` (conditional move) or similar will suffice.

This parsing stage transforms the expressions and statements within each symbol in-place so no new data structure is needed to report the new results. The final stage is quite different and is a lot closer to code generation than it really is to parsing.

`linearize_symbol()` takes a symbol that has been parsed, evaluated, and expanded; if that symbol represents a function it will produce a network of "basic blocks" represented by a single entry point. A basic block is a sequence of instructions with no jumps except at the end, so all jumps or control transfers are from the end of one basic block to the beginning of another. Performing the `EXPR_SELECT` optimization before this step can result in fewer basic blocks.

The details of this conversion and its usefulness are fairly impenetrable until you know about Static Single Assignment form (SSA), at which point they become relatively straightforward. The key elements of SSA are the basic blocks, the links between them representing jumps, and versioned variables. When a variable is assigned to multiple times in the code, SSA requires that variable be cloned, once per assignment, so that each final variable receives a single assignment. In sparse these versioned variables are referred to as `pseudos`.

Using SSA form simplifies a number of optimizations, including the dead-code removal in `if (0)` statements mentioned above. These optimizations are not only important when the aim is code generation, they are valuable for providing high-quality warnings, which is the main use case for sparse today. A simple example of this is the `__range__` statement that sparse adds to C. It is given three values such as:

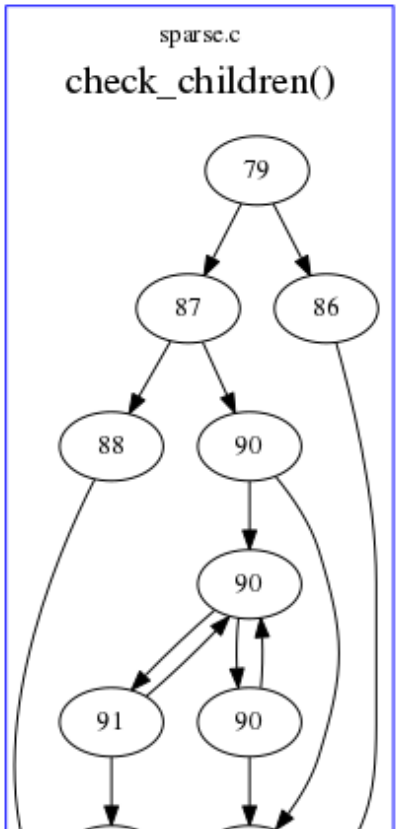
```
__range__ sizeof(struct foo), 0, 128
```

with the implication that the first value must be within the range given by other two. Once sparse has performed all the parsing steps and has a network of basic blocks, one of the tests it performs is to examine every instruction in every basic block and give the "value out of range" warning if the range-check operation (`OP_RANGE`) is present. This works because one of the late optimizations is to discard `OP_RANGE` instructions when all three values are constant and the first value is within the required range. In a language like C, where inline functions and macro expansion can place lots of dead code in unexpected places, it is important to remove as much of it as possible before passing judgment on that code's quality.

Use cases

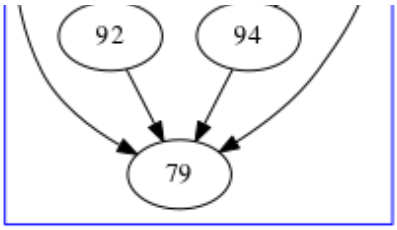
This window into the purpose and structure of the various interfaces of libsparse is focused on the miniature: the steps and details. As such it doesn't give much hint as to why anyone would care, or what sorts of tools can be built with it. Undoubtedly there are possibilities that haven't been implemented or even imagined yet, but a quick look at the tools that come with sparse might be useful for sparking new ideas.

Along with `test_inspect` which has already been mentioned and allows some elements of structure to be viewed, there is `graph` that converts the basic-block network into a graph description in `graphviz` format; see the example to the right. This, together with similar tools, can be helpful for students trying to understand how compilation works.



A data extraction tool of a different type is `ctags`. A "tags" file lists locations in a set of files that are particularly interesting, typically the locations of the definitions or declaration of different names in a program. Text editors can use these tags files to help the user navigate around the code. A tags file is typically generated by a fairly simple regular-expression-based parse of the file. While this this is often effective, it is not perfect. As

mentioned above, when a function is defined using a macro, a regular expression isn't going to be able to identify the function name, so the standard ctags and etags (an emacs-specific version) tools do not find such function definitions.



Sparse comes with a `ctags` tool that examines the symbol table generated during parsing of C code and creates a tags file recording exactly where every global symbol was defined, even when that was the result of multi-layered macro expansion. While `git grep` cannot tell me where the `__alloc_statement()` function was defined, the tags file created by `ctags` tells me it was on line 70 of allocate.h:

```
DECLARE_ALLOCATOR(statement);
```

If I can fit this `ctags` into my workflow I might need to find a new least favorite coding style.

The main tool that uses the `sparse` library is, of course, `sparse` itself, which reports various errors and warnings while examining the code. Some of these, such as the test of assignment in an `if` condition, are applicable to C in general, but a large class of the warnings that sparse generates come from extending the C language is various ways. Sparse defines the macro `__CHECKER__` so that the use of these extensions can be made visible only to `sparse`, not to other C compilers.

Some of these extensions, like `__range__`, are new statements, but most are attributes that can be attached to variable and type declarations using the GCC attribute syntax extension. These can provide extra information about how a variable or type should be used so sparse can warn when the expectations are not met. For example, there are two ways to initialize a structure, one of which is with positional initializers:

```
struct foo { int a,b; } positional = { 1, 2 };
```

The alternative is to use designated initializers:

```
struct foo { int a,b; } designated = { b:2, a:1 };
```

Positional initializers provide a simple list of values that are assigned to the fields of the structure based on their position in the list. Designated initializers, instead, attach the field name to each value in the initializer to avoid simple ordering errors. In 2009, sparse gained

support for a `designated-init` attribute that, when attached to a structure type, will trigger a warning if a positional initializer were ever used to initialize a structure of that type. This same attribute was copied to GCC in 2014 so we don't really need sparse any longer to get that warning, but there are other extensions that are less generally applicable and so less likely to make their way into GCC.

Two such extensions are related to the type system and have been discussed previously in these pages: `bitwise`, which creates a "new" type (in the Ada sense) that is identical to some other integer type except that it is incompatible with it, and `address_space`s, which provide similar functionality for pointers. `bitwise` can be used to avoid confusing big-endian and little-endian values, or to avoid accidentally using bitmasks on the wrong variable. The most obvious use of address spaces in the Linux kernel are to distinguish user-space pointers from kernel-space pointers, though there are other uses.

All of these extensions are amenable to simple static analysis: they enhance the type information in a way that allows certain operations to be easily seen as incorrect. Sometimes it would be nice to perform some more dynamic analysis, where a particular operation is valid only when proceeded or followed by some matching operation. A memory allocation must be followed by either releasing the memory or storing a reference somewhere, a pointer may only be dereferenced if it has been assigned a non-NULL value, a lock that has been taken must always be released, and so on.

The final SSA stage of sparse does allow for some dynamic analysis, but only at a very coarse level. It can often detect when a variable can be used without ever being assigned, but it cannot, for example, track if a variable is within a given range; that only works for constants.

One small step toward more general data flow analysis is found in the "context" tracking that sparse does to help catch errors where a lock is taken but not released. While the implementation is useful, it is extremely simplistic. It does not track individual locks at all but, instead, stores a single integer "context" counter for each basic block. Any "lock" event on any variable increments this counter, any unlock decrements it. As long as all paths through the code lead to the same context value at each location, it is assumed that the code is correct. This test would be easy to fool, but code designed to fool sparse would likely be quite obvious to humans, while a forgotten unlock calls on error paths, which humans may miss, would be obvious to sparse.

Building on sparse

This observation that sparse, while powerful, is sometimes simplistic is where we will leave sparse for now, though the interested reader is encouraged to explore and experiment with the code which is, of course, open. But this is not the end of our little foray into the internals of static analyzers. Smatch is a tool built on top of sparse which fills in some of the gaps left by sparse. If you have a desire to define some extensions to C to help catch more errors and sparse doesn't seem to be up to your task, smatch may be the tool for you.

Index entries for this article

Kernel	Development tools/Sparse
Kernel	sparse
GuestArticles	Brown, Neil

(Log in to post comments)

Sparse: a look under the hood

Posted Jun 9, 2016 9:25 UTC (Thu) by **ballombe** (subscriber, #9523) [Link]

Note that sparse 0.4.x was not licensed under the MIT license and is in the non-free section of Debian stable,
so this alone is a welcome development, see
https://packages.debian.org/jessie/sparse

Reply to this comment