Code Design and Documenting Considerations

Dick Grune
dick@dickgrune.com

September 21, 2014; DRAFT

1 Anti-aliasing

When a programming language has pointer parameters and global variables, there can be aliasing problems, one object being reachable under seemingly unrelated names.

In Teckel there are only three important global objects: `Root`, `Node`, and `Parent`. All of these have type `struct node *`. So there is a simple (heuristic) rule to prevent aliasing in Teckel:

> A routine can only refer to global objects when it has no `struct node *` parameters.

(There can still be aliasing among local variables, but that is much less likely to occur.)

There are two situations in which the test is not useful:

1. in the routine `_oops()`, which dumps information
2. in the file `print.c`, for the same reason

The program `check_aliases` implements these checks and exceptions.

Note that all this is very much geared to Teckel.

2 Tree Access

The postscan requires very frequent and intensive access to parts of the program tree. For example, the name of the first formal parameter in a function definition is reached by

```
fd->child[0]->child[1]->child[0]->child[0]->string
```

where `fd` is the node of the function definition.

There are 189 lines with `child[0]` or`child[1]` selections in the present code, representing a total of 232 choices. Such code is unpleasant to write and unmaintainable.

The answer to this kind of problem is naming: "sail your ship between Terschelling and Ameland" is much more informative than "sail your ship between the third and the fourth island".

The obvious way to do the naming is something like:

```
#define function_definition_2_function_definition_formal_part   child[0]
#define function_definition_formal_part_2_formal_parameter_list child[1]
#define formal_parameter_list_2_possibly_typed_identifier       child[0]
#define possibly_typed_identifier_2_identifier                  child[0]
```

```
fd->function_definition_2_function_definition_formal_part
->function_definition_formal_part_2_formal_parameter_list
->formal_parameter_list_2_possibly_typed_identifier
->possibly_typed_identifier_2_identifier
->string
```

which may not be easily readable (`_2_` is much more readable than `_to_`) but is at least checkable, and may be our best hope to avoid losing our bearings in the program tree. The `#define`s can be generated by `g2y`.

There is, however, one fly in the ointment: `g2y` knows about the parsing grammar (`syntax.g`) only, but at least two processes, the code generator and the postscan, restructure some parts of the program tree, thereby invalidating some `define`s and/or necessitating new ones.

There is a few things we can do about this, all pretty far-fetched:

- inventing a formalism for code generation and the tree transformations from which we can generate the pertinent code and the `define`s, Stratego style.
- use Stratego.

A poor man's version would be to access remote parts of the program tree as in the following example:

```
struct node *function_definition_formal_part =
    function_definition->child[0];
struct node *formal_parameter_list =
    function_definition_formal_part->child[1];
struct node *possibly_typed_identifier =
    formal_parameter_list->child[0];
struct node *identifier =
    possibly_typed_identifier->child[0];

identifier->string
```

This has the advantage that it does not require extensive modification of `g2y`, and that intermediate results can be reused. Its use implies, however, that after a modification of a node X in the program tree, all uses of `struct node *`X must be checked.

Each of the solutions emphasizes the importance of naming all nodes in the program tree.

3 Local Field Access

Some node types need to retain information from teckel run to teckel run. In an object-oriented language these would be implemented as classes. In this

project a field `void *info[N_INFO]` is set aside in each node for such data. The question is how to access such data in a node `n`.

- Simply as `n->info[0]`, `n->info[1]`, etc. One can then define field names: `#define Property info[0]` and write `n->Property = '2';`, etc.

  The problem is that the type of `n->info[0]`, etc. is always wrong, requiring ugly casts. One has to write `n->Property = (Void *)'2';`, etc.

- By imposing a fake structure over the `info` field.

  With the (fixed) macro definition

  ```
  #define LOCAL *local = (struct local *)(&(Node)->info[0])
  ```

  we can for example create a local struct definition based on the node called `Node`:

  ```
  struct local {
      char Property;
  } LOCAL;
  ```

  and access the field(s) thus:

  ```
  local->Property = '2';
  ```

  This has the advantage that it is quite readable and writable, but the disadvantages that 1. the name `local` is fixed and only one struct can be called `local` per scope, and that 2. the overlay is restricted to the node called `Node`.

- By more flexibly imposing a fake structure over the `info` field.

  With the (fixed) macro definition

  ```
  #define DATA(n) (&(n)->info[0])
  ```

  and for example the structure definition

  ```
  struct My_locals {
      char Property;
  };
  ```

  we can write

  ```
  struct My_locals *My_data = (struct My_locals *)DATA(That_node);
  ```

  to create a local struct definition on any node and access the field(s) thus:

  ```
  My_data->Property = '2';
  ```

  But this is a lot less readable and writable, and oh horror it allows constructions like:

```
(struct My_locals *)DATA(That_node)->Property = '2';
```

The basic problem is that the desired construction requires three names:

```
struct struct_name *local_name = (struct struct_name *)(&(node_name)->info[0]);
```

which is ugly and error-prone. To conform to C syntax we would like to write things like

```
struct struct_name *local_name = ...(node_name);              /* no macros */
```

rather than

```
Declare_Local(struct struct_name, local_name, node_name); /* one yuck macro */
```

Perhaps we should define a separate macro for each `struct_name`:

```
#define struct_name_locals(n) (struct struct_name *)(&(n)->info[0])
```

which would allow us to define

```
struct struct_name *local_name = struct_name_locals(node_name);/* one macro */
```

for any node and which looks pretty acceptable.

3.1 Samples

```
/* delay_token */
struct delay {
int delay;
};

if (Node->status == NEW) {
((struct delay *)DATA(Node))->delay = 10;
Node->status = WORK;
bye();
return;
}

/* WORK */
int cnt = ((struct delay *)DATA(Node))->delay--;
if (cnt) {
((struct delay *)DATA(Node))->delay = cnt;
return;
}
```

4 Minor Issues

4.1 Modules

Teckel consists of modules, each consisting of a header file (`.h`) and a code file (`.c`).

The header file contains declarations for all items the code file defines, plus all items provided by the module through `#define`s, f.e. `#define True (1)` in `Bool.h`.

Fine-tuning parameters of a module go into the code file, f.e. `#define PAGE_WIDTH 80` in `print.c`.

The proper forms of the modules is verified by calling the (non-trivial) Shell script `check_ch`.

The `syntax.tab` module is generated by *bison* and does not conform.

4.2 Header files including header files

In general header files should not include header files. If a header file `H.h` uses a data type `T` that is defined in `T.h`, any code file that uses `H.h` almost certainly also uses the type `T` and might as well explicitly include `T.h`, preferably before `H.h`.

But there are exceptions. One is `node.h` ZZ
`CRC.h` ZZ

4.3 Temporary string buffers

The module `memory.c` provided a routine `new_string_buffer(void)` which yielded a 256-byte transient buffer from a circular list of 20 buffers. Having been bitten by such a string being passed as a parameter to a routine that used up those 20 buffers and then used the parameter, the routine is now named `temp_string_buffer(void)`, and all routines that yield transient strings have the syllable `temp_` in them.

This has uncovered two further problem spots. The problematic transient strings are now allocated through `permanent_string()`.

It would be possible to mark transient strings by some special type, for example pointer to unsigned char. This would get us compiler help in using transient strings consistently, but would probably require a lot of conversion calls.

4.4 How to concatenate with `sprintf()`

The routine `printf()` is very convenient for printing formatted output; the routine `sprintf()` is fat less convenient for producing formatted strings. The reason is that the printer (whatever that is) concateretes the successive results of the calls of `printf()`, but no such automatic mechanism is provided by `sprintf()`

Two tricks suggest themselves. After

```
sprintf(buf, "text, %d", 42);
```

we can continue with

```
sprintf(buf + strlen(buf), ", next, %d", 24);
```

or with

```
sprintf(buf, "%s, next, %d", buf, 24);
```

to have `printf("%s\n", buf)` produce the concatenated result

```
text, 42, next, 24
```

Although the second method seems to work, it is scary, and it is probably safer to stick with the first method.

4.5 Down with `while`

When writing a while statement it is easy to forget the statement to move to the next item. Now this is not an error that easily goes unnoticed, but it is a nuisance. It is probably better to use a for-statement wherever possible:

```
struct node *n;
for (n = start; n; n = RC(n)) {
```

rather than

```
struct node *n = start;
while (n) {
    ...
    ...
    n = RC(n);
}
```

5 Documentation

- The presence of `\_` in the documentation is a nuisance because it makes it harder to find identifier and file names. But changing them into `\verb@@` constructions does not work since there are quite a lot of places where `\verb` is not allowed. We therefore opt for the other extreme and do use `\verb` to hide a `_`.

  This means that to find names with `_` in the documentation, one has to search for `\_`. The search scripts `F_tex` and `F_all` do that automatically.