Gont for C programmers

February 20, 2002

`$Id: c2gont.tex,v 1.7 2002/02/19 09:20:21 malekith Exp $`

1 What Gont is meant to be?

Largely imperative composition of C-like lexical layer, control structures and speed with ML's typesystem, functions as first class citizens, and safety. Plus possibly few more things, like objects.

What is Gont different from C? C is referred to as a C89 ANSI standard (see K&R's The C Language, second edition). Notes about implementation are given in [[]].

2 General, conceptual, differences

2.1 Safety

Gont is safe. This means that compiler shouldn't produce any code, execution of which would cause SEGV. Parse: it should not be possible to overflow buffers, mismatch types of arguments in calls and so on. However note, that SEGV can be obtained from playing with `extern_c` declaration, compiler has no means of validating.

2.2 Level of abstraction

Gont is much higher level. It generally operates at much higher level of abstraction.

<evangelisation>

> [...] And, actually, the more you can avoid programming in C the more productive you will be.
>
> C is very efficient, and very sparing of your machine's resources. Unfortunately, C gets that efficiency by requiring you to do a lot of low-level management of resources (like memory) by hand. All that low-level code is complex and bug-prone, and will soak up huge amounts of your time on debugging. With today's machines as powerful as they are, this is usually a bad tradeoff – it's smarter to use a language that uses the machine's time less efficiently, but your time much *more* efficiently. Thus, Python.
>
> Eric S. Raymond

This is quotation from hacker howto by ESR :-). One may easily note that it does not only talk about Python (personally I don't like dynamic typing nor using interpreted language as a general purpose, –mal).

C is, in principle, assembly language, therefore it is almost as fast as real assembly, but it is still easy to write and quite portable. It is great for programming things like kernels, libraries that need high speed operation (like MPEG decoding/encoding), generally most of things that need to be real time. However it is very easy to cut your finger with such a sharp knife. Typesystem

provided by C isn't very powerful, most of people familiar with functional programming, would say that typesystem of C is no more then a toy. Similarly C doesn't provide much support for memory management, exceptions, functional and objective programming. Of course, lack of restriction from language, about how you organize your memory management, objects, and error recovery, can be advantage, especially with respect to operating system programming and so on. However in most cases, writing `xmalloc()` 100th time, is annoying. I think that's the main reason why Java is so popular – it takes burden of thinking at the very low level away from the programmer, and also it is much harder to make a mistake (at the cost of more keyboards wasted on writing `public virtual final void foo()` and so on, and performance regression).

</evangelisation>

2.3 Memory Management

Gont provides transparent memory management, along with garbage collection (the particular kind of garbage collector used is Boehm conservative garbage collector, used also by GCJ and Popcorn). It generally means you can allocate as much as you want, forget the pointers, and the Gont runtime system with get rid of unused memory.

2.4 C compatibility

Gont is not compatible with C (this is difference more against C++ then C). It is simply impossible to maintain such compatibility in safe language. What a braindamage might result, one may easy tell from the C++ example.

2.5 Differences against ML

How is Gont different from Caml or SML? (Caml and SML are functional languages, with imperative features from ML family) Hm... generally all languages are interchangeable, what can be written in one, can be written in all other. However in real life it is rather important how easy can you get the code to work, how much bugs will compiler detect (vs bugs left for the programmer) and how fast will it run. Gont places accents on these things somewhere between Caml and C. Generally it does not provide as much support for functional programming as Caml does, similar can be told about Gont's module system (which is a toy, compared to functors and other ML machinery) and restricted polimorphism. On the other hand, linking Gont code with C is very easy, the only thing you need to remember, is not to put pointers from Gont, in `malloc()`'ed area – save it on stack, or in `GC_malloc()`'ed area. Interfacing OCaml is... ghm... nightmare, mainly because of its precise garbage collector. Also Gont code will probably run faster, as it uses highly optimizing back end (gcc), and because of restrictions put on the language itself (this is probably not true yet).

2.6 Differences against Popcorn

How is Gont different from Popcorn? (Popcorn is safe C subset, compiler is available to TALx86 (Typed Assembly Language)). Popcorn was inspiration for Gont :) However, it is somewhat limited (especially to x86) and not currently under development (AFAIK).

2.7 Differences against Cyclone

Cyclone (`http://www.cs.cornell.edu/projects/cyclone/`) is language similar to Gont in the same sense as C++ is similar to Java. It's a dialect of C designed to be safe: free of crashes, buffer overflows, format string attacks, and so on. Cyclone has more powerful typesystem – it includes regions, that allow not to use garbage collection all the time. It also has pointers in C's sense.

OTOH Gont is not dialect of C. It does not try to be backward compatible. This results in much smaller language, probably easier to understand at first.

3 Control structures

3.1 Empty instruction

There is no empty instruction (written as ';' in C). `skip` keyword is introduced instead. For example:

C:

```
while (f(b++));
```

Gont:

```
while (f(b++))
        skip;
```

This is to disallow common C's mistake:

```
while (cond);
        do_sth;
```

(note ';' after while).

3.2 Control structures

- `if` *cond stmt*
- `if` *cond block* `else` *block*
- `while` *cond stmt*
- `do` *stmt* `while` *cond*

cond is expression of type bool. (therefore `int i=10; while (i--) f();` is not allowed, one needs to write `while (i-- != 0)`). *stmt* might be single statement or a *block*. *block* is zero or more *stmt*'s surrounded with `{ }`. One might note that if's version with `else` keyword is required to have `{ }` around each part. This is to solve dangling-else problem, for example:

C:

```
if (b1)
        if (b2)
                f1();
else
        f2();
```

This if of course parsed as:

```
if (b1) {
        if (b2) {
                f1();
        } else {
                f2();
        }
}
```

In C `else` is associated with nearest else-free `if`. As you have seen this can be source of problems.

However, as there is no danger with using `if` without `{ }` and `else`, (as in `if (x == 0) x++;`), it can be used without `{ }`. It is also allowed to have `if (x == 0) x++; else x--;`, as there are no `if` in `x--` nor `x++`.

3.3 Labeled loops

In Gont, similarly to Ada or Perl, loops can be given names, in order to later on tell `break` and `continue` which exactly loop to break. This looks as:

```
foo: while (cond) {
        bar: for (;;) {
                if (c1)
                        break foo;      // break outer loop
                else
                        continue bar;   // continue inner loop
                for (;;) {
                        if (c2)
                                break;  // break enclosing loop
                }
        }
}
```

4 Data types

4.1 Pointers

There is no explicit notation of pointers. Structures and unions are always treated as pointers (as in Java). Strings are built in type. Arrays are implemented as abstract type. Pointers to functions are discussed below.

4.2 Structures

They are defined with `struct` and `opt_struct` keywords.

```
struct s {
        int a;
        string b;
};
```

is roughly equivalent of C's:

```
typedef struct {
        int a;
        char *b;
} *s;
```

So, name `s` can be used (only) without any `struct` or `opt_struct`, as type. You should note, that structures are passed by pointer (or by reference if you prefer C++ naming style), therefore changes made to struct inside function are reflected in state of it on the caller side.

You can access fields of structure with '.' operator.

Now, what `opt_struct` is all about. `struct` value is always valid, i.e. it has to be initialized with object instance, and you cannot assign `null` pointer to it (BTW: `null` is keyword). `opt_struct`'s are not. They can be initialized with `null`, as well as assigned `null`. This involves runtime check on each access to `opt_struct` value. When you try to access `opt_struct` value, that is `null`, `Null_access` exception is raised.

If you do:

```
void f(s1 x)
{
        s1 y;
        y = x;
        y.fld = 0;  // here you also modify x.fld
}
```

In general assigning values other then `int`'s, `bool`'s and `float`'s copies pointer, not content, i.e. makes an alias of object.

4.3 Structure initializers

When initializing structures one has to spell field name along with expression initializing it. For example:

```
struct foo {
        int bar;
        string baz;
}

void f()
{
        foo qux = ({ bar = 1, baz = "quxx" });
}
```

`({ ... })` is also regular expression, for instance:

```
void f()
{
        foo(({ bar = 1, baz = "qux" }));
}
```

5 Features of Gont typesystem

5.1 Functional values

Functions are first class citizens. This functional programming slogan means that you can write a function, that takes function as argument and returns function. For example:

```
opt_struct int_list {
        int data;
        int_list next;
}

*(int_list) -> void mapper(*(int) -> void f)
{
```

```
        void for_all(int_list lst)
        {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        }

        return for_all;
}
```

This function takes function `f` operating on items, and returns function that applies `f` to all elements of list.

The key thing to note is functional type notation. `*(int) -> void f` means roughly `void (*f)(int)` – so it is function that takes single int parameter and returns no value. Similarly `*(int, int_list) -> string` is function, that takes `int` and `int_list` parameter, and returns `string`.

`int_list` is passed to functions as pointer (in C's sense).

Functional values are not simply passed as pointers to functions. That wouldn't work for `for_all` function from our example, since it needs `f` parameter that would disappear from the stack after `mapper` has returned. So functional values are passed as pairs of pointers to functions, and pointers to special closure structures.

Nested function definitions (for example the `for_all` function) are shortcut to defining functional variables and initializing them, so our example could look as:

```
*(int_list) -> void mapper(*(int) -> void f)
{
        *(int_list) -> void for_all;

        for_all = fun (int_list lst) -> void is {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        };

        return for_all;
}
```

Or even:

```
*(int_list) -> void mapper(*(int) -> void f)
{
        return fun (int_list lst) -> void is {
                while (lst != null) {
                        f(lst.data);
                        lst = lst.next;
                }
        };
}
```

Special extension is supported, useful when dealing with functional values. Whenever function body (sequence of statements within `{ }`) should appear, single expression within `( )` can be supplied. It is passed to return statement, which is only statement in function body. So:

```
fun (int a, int b) -> int is (a + b)
```

is equivalent of

```
fun (int a, int b) -> int is { return a + b; }
```

5.2 Tuples

Tuple is datatype, that consists of two or more components, which are anonymous. Tuples are defined as:

```
*[int, int] t1;
*[int, string, bool] t2;
*[ctx, ty] t3;
```

And used as:

```
t1 = [1, 2];
int a, b;
[a, b] = t;
// swap
[a, b] = [b, a];
t3 = [1, "hello", true];
// true and false are keywords
```

More on tuples in 'Patterns' below.

5.3 Unions

Union are more closely related to ML's datatypes then to C's unions. The basic difference is that Gont compiler remembers which member is currently stored in union.

Unions are defined as:

```
union exp {
        int Const;
        string Var;
        *[exp, exp] Add;
        *[exp, exp] Sub;
        *[exp, exp] Mul;
        *[exp, exp] Div;
}
```

This union can be later on used for processing symbolic expressions. You can access union components only using pattern matching (there is no '.' notation).

6 Polimorphism

This is probably the neatest thing in Gont. Structures, as well as functions can be parameterized over types. For example to define list of anything you write:

```
opt_struct <'a>list {
        'a data;
        <'a>list next;
}
```

'a is alpha. It is type variable, it stands for any type (similarly 'b is beta, but I really don't know what 'foo is... :-)

Then you use it as:

```
<int>list l;            // list of ints
<<int>list>list ll;     // list of lists of ints
```

If you are familiar with C++ you can note '«' that has to be written as '< <' there. This is not the case in Gont. It is handled specially.

Functions can be written that operate on lists:

```
// Call passed function f on each element of the list l
// starting from head.
void iter(*('a) -> void f, <'a>list l)
{
        while (l != null) {
                f(l);
                l = l.next;
        }
}

// Call function f on each element of the list l,
// collect results as a list (of possibly different type)
// and return it.
<'b>list map(*('a) -> 'b f, <'a>list l)
{
        <'b>list o = null;

        while (l != null) {
                o = {data = f(l.data), next = o};
                l = l.next;
        }

        return o;
}
```

Later on you can use defined functions.

Suppose you have defined:

```
string int2string(int);
void print_string(string);
```

somewhere, then:

```
<int>list il = { data = 1, next = { data = 2, next = null }};
<string>list sl = map(int2string, il);
iter(print_string, sl);
```

7 Module system

Modules provide way of separate compilation along with possibility of avoiding namespace conflicts. Current module system is based on ideas from OCaml. However it is very limited, we only support one level of modules, there are no functors and so on. Module Foo consist of two files: foo.g

(implementation) and foo.gi (interface). Whatever names are placed in foo.* files, they are all prefixed with 'Foo::'. gontc compiles foo.gi file to foo.gio, and foo.g to foo.o. foo.gio is needed whenever you access Foo:: symbols from other modules.

Example:

list.gi:

```
// this is to output information,
// that we implement type 't', but not to
// disclosure what it is.
type <'a>t;

// return first element (head) of the list
'a hd(<'a>t x);

// return all but first element (tail) of the list
<'a>t tl(<'a>t x);

// some bogus, random variable -- you can export variables
// the same way as you would with functions
int cnt;

// create new empty list
<'a>t create();

// apply f to all elements of l, return list of results
<'b>t map(*('a) -> 'b f, <'a>t l);

// call f on all elements of the list
void iter(*('a) -> void f, <'a>t l);
```

list.g:

```
// this is local datatype.
opt_struct <'a>t {
        'a data;
        <'a>opt_struct next;
}

// this will be exported out ('public')
'a hd(<'a>t x) { return x.data; }
<'a>t tl(<'a>t x) { return x.next; }

// this is local, can't be called from outside the module
void helper(<'a>t x) { ... }

// and more publics
int cnt;
<'a>t create() { cnt++; return null; }
<'b>t map(*('a) -> 'b f, <'a>t l) { ... }
void iter(*('a) -> void f, <'a>t l) { ... }
// ...
```

Then if you want to use the module, it can be done with :: notation, like this:

```
<int>List::t l = List::create();
...
int k = List::hd(l);
```

In case of some modules it might be useful to open them, i.e. import all symbols from module intro current namespace, so you no longer have to use :: notation (but you still can, it is often suggested for readability):

```
open List;
...
<int>List::t l = List::create();
...
int k = hd(l);
<int>List rest = tl(l);
```

8 Pattern matching

It is especially useful with conjunction with tuples and unions. Patterns are used in switch and let statements, like this:

```
int compute(exp e)
{
        switch e {
        case Const[x]    : return x;
        case Var[x]      : return lookup_var(x);
        case Add[e1, e2] : return compute(e1) + compute(e2);
        case Mul[e1, e2] : return compute(e1) * compute(e2);
        case Div[e1, e2] : return compute(e1) / compute(e2);
        case Sub[e1, e2] : return compute(e1) - compute(e2);
        }
}

exp diff(exp e)
{
        switch e {
        case Const[x]    : return Const[0];
        case Var[x]      : return Const[1];
        case Add[e1, e2] : return Add[diff(e1), diff(e2)];
        case Sub[e1, e2] : return Sub[diff(e1), diff(e2)];
        case Div[e1, e2] :
                exp up = Sub[Mul[diff(e1), e2], Mul[e1, diff(e2])];
                exp down = Mul[e2, e2];
                return Div[up, down];
        case Mul[e1, e2] :
                return Add[Mul[diff(e1), e2], Mul[e1, diff(e2])];
        }
}
```

To convince you, we're not so far from C:

```
union color {
        void Red;
        void Green;
```

```
        void Blue;
}
```

This is roughly equivalent of C's:

```
typedef enum {
        Red,
        Green,
        Blue } color
```

Then we do:

```
int spy007;

string color_name(color c)
{
        string r;

        switch (c) {
        case Red:
                spy007++;
                r = "red";
        case Green:
                r = "green";
        // [] also allowed
        case Blue[]:
                r = "blue";
        }

        return r;
}
```

You can note lack of break at the end of case clauses. They are not required, as there is *no fall through* (because case clause has to introduce new scope).

In previous examples we have checked for each possibility, but we don't have to:

```
string var_name1(exp e)
{
        switch e {
        case Var[x]: return x;
        // matches any x
        case x: return "not variable";
        }
}

string var_name2(exp e)
{
        switch e {
        case Var[x]: return x;
        }
}
```

`var_name2` would raise `Match_failure` exception if any pattern didn't match.

Patterns can be used to decomposite tuples, like this:

```
*[int, int] t = [1, 2];

...
switch t {
case [i1, i2]: return i1 + i2;
}
```

This can be abbreviated to:

```
let [i1, i2] = t in { return i1 + i2; }
```

There can be more then one assignment, like this:

```
let [t1, t1] = t,
    [s1, s2] = s {
        // ...
}
```

The let assignment and binding names with case just creates new name for an object. Specificly it means that assigning values to names bound with let/case changes object itself. Example:

```
*[int, string] t = [1, "one"];

switch t {
case [i, s]: i = 2;
}

let [i, s] = t in { s = "two"; }

// here t == [2, "two"]
```

One can note that you can also decomposite t with:

```
string s;
int i;
[i, s] = t;
// here i = 2, s = "two"
// however:
i = 3; s = "three";
// here i = 3, s = "three", but t == [2, "two"]
```

You can also pattern-match structures [[you cannot yet :-]]

9 Exceptions

They are used to inform about unusual situation in a program, in order to transfer control to block, that can handle it. They can be thought of as member of a huge union:

```
union exn {
        // system defined
        void Null_access;
        void Match_failure;
        void Not_found;
```

```
        // user defined, Compiler is name of user module
        string Compiler::Syntax_error;
        void Compiler::ICE;
}
```

New members are added with exception keyword:

```
exception string Syntax_error;
exception void ICE;
```

In order to signal unusual situation, you use raise keyword:

```
raise Syntax_error["parse error"];
raise ICE;
raise Compiler::ICE[];
```

At the place, where you know how to handle it, you use try:

```
try {
        open_file();
        ...
        parse();
        ...
} with e {
case Syntax_error[s]:
        print_msg(s);
case ICE:
        print_msg("Internal Compiler Error");
} finally {
        close_file();
}
```

First we open some file, then we call `parse()`, that can raise `Syntax_error` or `ICE` in which case we print error message, or something else, but in then control is transfered to upper-level try block. No matter how control leaves try {} block instructions in finally { ... } are executed. So we always close the file.

[[Nope... but they are expected right after modules, which means Real Soon Now]]

10 Constructing functional values

There are two ways to build functional value in Gont. First comes from C:

```
int add(int a, int b) { return a + b; }
```

and the second from ML:

```
*(int, int) -> int add =
        fun (int a, int b) -> int is { return a + b; };
```

The second way might seem odd to C programmer. Of course, because it is odd when used to define simple C-like function. But it is not when you need to pass function to another function, let's say:

```
void fputs(file f, string s);
...
void do_sth(int foo, *(string) -> void err_report);
...
do_sth(16, fun (string s) -> void is { fputs(f, s); } );
```

Oh well... it's probably still odd ;)

People familiar with ML might recognize that despite the type information for `fun (...)` statement can be obtained from itself, it still has to be given (even twice). I guess it is place for experiments.

I also would like to have some support for named arguments, so functions can be called as:

```
Window::create(width => 20, height => 23, color => red);
```

This should come with default values.

11 Typesystem

t, t1, t2, ... are type expressions. v1, v2, ... are type variables ('a, 'b, 'foo, ...).

Basic types: `int`, `float`, `string`, `bool`, `void`, written as is.

Function type is written as follows:

```
*(t1, t2, ..., tn) -> t
```

where n >= 0. One might note that this is somewhat different from ML where all functions take just one parameter (which can be tuple or another function, but this is not the point). This is closer to C's function notation.

Tuples:

```
*[t1, t2, ..., tn]
```

where n >= 2.

Structures are defined with:

```
struct <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

Structures that can be invalid (i.e. null):

```
opt_struct <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

Unions (datatypes) are defined with:

```
union <v1, v2, ..., vn> NAME {
        t1 f1;
        t2 f2;
        ...
        tm fm;
}
```

where n >= 0, m >= 1. If n == 0, <> can be omitted. fi are field names.
Structures and unions can be later on referred with:

```
<t1, t2, ..., tn> NAME
```

where n >= 0. If n == 0, <> can be omitted.

* in front of tuple and function types is (crude) way to convince ocamlyacc (Bison has the same problem, though), that there are none reduce/reduce conflicts in grammar. I guess there aren't even without '*', but tell it to yacc... (this problem is referred to in Bison manual, there are some workarounds, however I was unable to make use of it).