Implementation Notes (aka hacker guide)

May 28, 2002

$Id: hacker.tex,v 1.7 2002/05/27 15:08:38 malekith Exp $

Contents

1 Intro

The compiler is written in OCaml. This ensures high portability, reasonable speed, along with modularization and type safety.

2 Coding conventions

Comment much. I know, there are not too much comments in the code at the very moment, but please forgive me, I'm working on it.

Leave code clean and warning free. This is especially true for OCaml warnings about pattern matching. The usual way of adding new things, is to add them to datatypes, and x match'es, until you'll get warning-free compilation :-)

Code should t 80 column screen.

2.1 For OCaml code

Basic indentation level is 2. Set expandtab option in your editor, so we have no **\t** in source les.

It is often to list all cases in patter matching, like:

```
match t with
| NamedType r -> NamedType {r with nt_module = m}
| Int | Void | Bool | Float | String | TyVar _ | AbstractType
| Tuple _ | Func _ | Struct _ | OptStruct _ | Union _ -> t
```

while one could have used:

```
match t with
| NamedType r -> NamedType {r with nt_module = m}
| _ -> t
```

which would be shorter, and probably more readable. However, after we have added new type, we should check each place, where we use pattern matching on type to see if it needs special case. OCaml will warn you if you add new type in the rst case (it is unmatched), in the second { it won't.

Never use **==**, **!=** nor **List.memq** for equality between datatypes:

```
let x = Foo in
let y = Foo in
(x == y, x = y)
```

value of this can be **(true, true)** or **(false, true)**.

Always use **==**, **!=** and **List.memq** for equality between objects of symbol type (cycles!).

2.2 For C code

indent -kr -i8, parse: K&R, tabstops are 8 spaces long. Don't use expandtab, we want **\t**'s here.

Example (very stupid one):

```
int f(int a, int b)
{
        while (a != 0) {
                if (a == b) {
```

```
                        a = b;
                } else {
                        b = c;
                }
                switch (c) {
                case C1:
                        c = 1;
                        break;
                case C2:
                        c = 2;
                        break;
                }
        }
}
```

Functions should be small (1-2 screens).

Why -kr? Because (1) K&R are right, (2) K&R are right, and (3) K&R are right. Why -i8? After 15th hour of hacking, you'll surly nd out why... If you get far too far to the right with such a huge indent, it means your functions are far too big (borrowed from CodingStyle le that comes with linux kernel).

3 Code generation

It is done in module Gontcodegen.

Compiler outputs intermediate language, called Ksi, that is later on compiled by gcc front end. Ksi itself is rather strike through interface to trees used by gcc internally. It uses lispy syntax.

Ksi output is done through Ksi module. It exports datatype for Ksi expressions and declaration, and function to convert this data into output stream (that is indented, to allow easier debugging of compiler output).

4 Argument passing conventions

Values passed as polymorphic parameters has to be all of the same size. Otherwise we wouldn't be able to generate e cient code. Specially for this purpose 'int' Gont type is chosen with the same as size as machine pointer. (This is not the choice of some C compilers on some 64 bit architectures, for example on alpha-*-linux with GCC sizeof(int) = 4, sizeof(long) = sizeof(void*) = 8).

They are few basic types of passing arguments in Gont.

1. by value. int and bool are passed this way.
2. by pointer to some location on the heap. structures, unions and tuples are passed this way.
3. by pointer to location, that is not guaranteed to exists after function terminates (read: this can be pointer to the stack of function that called us). If such value needs to be saved after function return, is has to be copied to the heap. It also has to be copied if calling raise with it. Special functions are provided by the runtime to copy such values. Floats and functional values are passed this way.

[[Functional values are currently always passed as pointers to the heap. I guess some semantic analysis will be done to pass it as pointers to stack when it is safe. Similarly for function closures.]]

Functional values cannot be passed simply as pointers to functions, since they might need closure in some (most?) cases. Here is layout of data structure describing functional value:

```
void *fnc;
void *closure;
```

closure has to be passed as the first argument to fnc. Closure must not point into the stack. Pseudocode example:

```
foo()
{
        int x;
        int y;
        g()
        {
                int p;

                use(p);
                use(x);
        }

        use(x);
        use(y);
        use(g);
}
```

is translated to:

```
struct foo_closure {
        int x;
}

foo_lambda1(foo_closure *ptr)
{
        int p;
        use(p);
        use(ptr->x);
}

foo()
{
        int y;
        foo_closure *cl = make_closure(...);
        void *g = make_lambda(foo_lambda1, cl);

        use(cl->x);
        use(y);
        use(g);
}
```

(Examples are given using pseudo-Gont as input, and pseudo-C as output, just to not confuse reader with Ksi syntax).

One might note that it might be impossible to pass regular function this way. We need auxiliary function to be generated.

5 Datatypes representation

ptro is signed integer type of size of pointer.

```
struct lambda_exp { void *fnc, *closure; }
```

5.1 Basic datatypes

int and **bool** is represented on stack and passed as **ptroff**. **float** is passed around as **double***, on stack it's simply **double**. **void** { **(void*)whatever**, if it needs to be passed, 'whatever' cannot be relied upon to contain anything speci c :^). ***(... (...))** { **struct lambda_exp***.

5.2 Strings

```
struct string {
        int len;
        char *data;
}
```

len is length of string, in characters. data is string content, it does not have to be NUL terminated. Strings are passed as struct string*

5.3 Tuples

[t1, ..., tn] is represented as pointer to array containing tuple content. All ti are treated, as if they where passed to function, i.e. sizeof(ti) == sizeof(void).

```
f(*['a,int,'b] x) {
        let (_, y, _) = x {use(y);}
}
```

is translated to:

```
f(void **x) {
        use((ptroff)x[1])
}
```

5.4 Unions

```
struct union_exp {
        ptroff sel;
        void *data;
};
```

Values of sel are assigned sequentially from 1, in order they are given in union de nition. Union values are passed between functions are struct union_exp*

5.5 Structures

They are treated much like C structures. They are always passed as pointers to structures. Similarly, if one structure contains another { it contains pointer to it, not it directly, thus:

```
struct foo {
        int x;
        string s;
}

struct bar {
        foo f;
        int z;
}
```

is translated to:

```
struct foo {
        int x;
        struct string *s;
};

struct bar {
        struct foo *f;
        int z;
};
```

5.6 By value or by reference

Ints, oats and bools are passed by value. For oats it will require sth like this:

```
float x,y,r;
...
r = tan2(x,y*2);
```

to be translated to:

```
{
        double _1, _2;
        _1 = x;
        _2 = y*2;
        r = *(double*)tan2(&_1,&_2);
}
```

Value returned from tan2() has to be stored in GC_malloc'ed() area, otherwise it typeof(tan2) would be not a subtype of *('a ('a,'a)).

Other datatypes (unions, structures, tuples and objects) are passed by reference.

There can be confusion with:

```
f(*[int,int] a)
{
        let [x,y] = a in {
                x = 10;
                y = 20
        }
}

g()
{
```

```
        int x, y;

        x = 5;
        f([x,y]);
        // whatever here x is still 5, or 10?
}
```

It is 5. However this more due to implementation, then to anything else, because it is inconsitent with:

```
[x, y] = [10, 20];
```

after which x is 10 and y is 20. But this is general problem with treating variables as addresses on the left side of '='.