Lowering high-level language constructs to LLVM IR

Ramkumar Ramachandra

October 20, 2014

it's pretty readable

```
1  define %value_t* @0() {
2  entry:
3    %value = call i8* @malloc(i64 ptrtoint (%value_t* ↩
         getelementptr (%value_t* null, i32 1) to i64))
4    %malloc_value = bitcast i8* %value to %value_t*
```

```
1    switch i32 %load19, label %default [
2      i32 1, label %caseN
3      i32 4, label %caseN24
4    ]
5
6  default:                              ; preds = %entry
7    %load21 = load i1* %boxptr10
8    br label %switchcont
9
10 caseN:                                ; preds = %entry
11   %boxptr22 = getelementptr inbounds %value_t* ↩
         %malloc_value8, i32 0, i32 1
12   %load23 = load i64* %boxptr22
13   %intbool = icmp eq i64 %load23, 0
14   %. = select i1 %intbool, i1 false, i1 true
15   br label %switchcont
```

the simplest program

```c
int add(int x, int y) {
    return x + y;
}
```

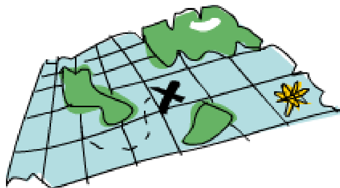
```llvm
define i32 @add(i32 %x, i32 %y) {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  store i32 %y, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

a few interesting instructions

```
1  #include <stdio.h>
2
3  int add(int x, int y) {
4      return x + y;
5  }
6
7  int main() {
8      printf("%d", add(3, 4));
9  }
```

```
1  @.str = private unnamed_addr constant [3 x i8] c"%d\00"
2
3  define i32 @main() {
4    %1 = call i32 @add(i32 3, i32 4)
5    %2 = call i32 (i8*, ...)* @printf(i8* getelementptr ↩
         inbounds ([3 x i8]* @.str, i32 0, i32 0), i32 %1)
6    ret i32 0
7  }
8
9  declare i32 @printf(i8*, ...)
```

map!

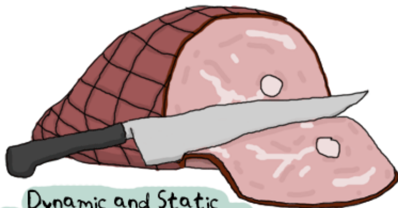


```
map :: (a -> b) -> [a] -> [b]
```

the functor

```
1 ["foo" "bar" "quux"]
2 [1 3 5 8]
3 [2.1 4.3 3.5]
4 [true false false true]
5 [true 4.3 8 "quux"]
```

types dance




```
i64
double
i1
i8
i8*
```

sum type (tagged union?)

```
1 %value_t = type {
2       i32,
3       i64,
4       double,
5       i1,
6       i8,
7       i8*,
8 }
```

the array type

```
1  %value_t = type {
2        i32,
3        i64,
4        double,
5        i1,
6        i8,
7        i8*,
8        %value_t**,                                    ; array
9        i64,                                           ; array length
10 }
```

a look at the implementation

```
1  (defn map
2    [f coll]
3    (if coll
4      (cons (f (first coll))
5            (map f (rest coll)))
6      []))
7
8  (defn map2
9    [f c1 c2]
10   (if (and c1 c2)
11     (cons (f (first c1) (first c2))
12           (map2 f (rest c1) (rest c2)))
13     []))
```

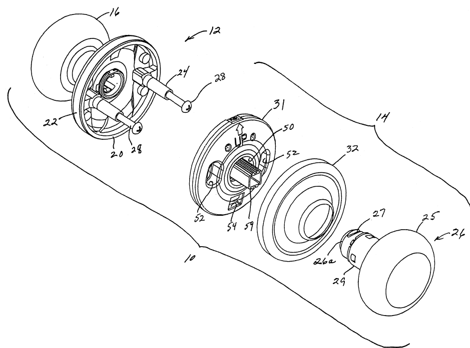
the function type

```
1  %value_t = type {
2       i32,                               ; type of data
3       i64,                               ; integer
4       double,                            ; double
5       i1,                                ; bool
6       i8,                                ; char
7       i8*,                               ; string
8       %value_t**,                        ; array
9       i64,                               ; array length
10      %value_t* (i32, ...)*,             ; function (simplified)
11 }
```

a vaargs refresher

```
1  #include <stdarg.h>
2
3  int vaargf(int nr, ...) {
4      va_list ap;
5      int ret;
6
7      va_start(ap, nr);
8      ret = va_arg(ap, int);
9      va_end(ap);
10     return ret;
11 }
```

varargs in llvm



`@llvm.va_start`

`@llvm.va_end`

~~`@llvm.va_arg`~~

the x86 detail

```
%struct.__va_list_tag = type { i32, i32, i8*, i8* }
```

```
1  let build_va_arg_x86 ap argtype =
2    let el = build_alloca argtype "el" builder in
3    let idxptr = build_gep ap (idx 0) "idxptr" builder in
4    let idx0 = build_load idxptr "idx" builder in
5    let magic_lim = const_int i32_type 40 in
6    let elsptr = build_gep ap (idx 3) "elsptr" builder in
7    let els = build_load elsptr "els" builder in
8    let rawel = build_gep els [| idx0 |] "rawel" builder in
9    let elptr = build_bitcast rawel (pointer_type argtype) ↩
         "elptr" builder in
10   let newidx = build_add idx0 const_8 "newidx" builder in
11   ignore (build_store newidx idxptr builder);
12   let newval = build_load elptr "newval" builder in
13   ignore (build_store newval el builder);
14   build_load el "ret" builder
```

one step further

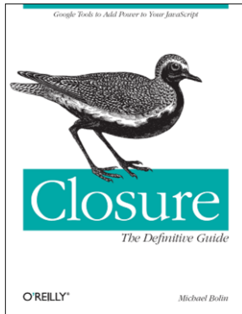
```
1  (defn add2 [a b] (+ a b))
2  (defn sub2 [a b] (- a b))
3
4  (defn map2
5    [f c1 c2]
6    (if (and c1 c2)
7      (cons (f (first c1) (first c2))
8            (map2 f (rest c1) (rest c2)))
9      []))
10
11 (def map2f add2)
12
13 (defn main []
14   (let [map2-wrapper (fn [c1 c2] (map2 map2f c1 c2))
15         map2f sub2]
16     (map2-wrapper [4 5 6] [1 2 3])))
```

λ lifting

```
1 (defn lambda-wrapper []
2   (fn []
3     (println aenv)
4     (println env)))
```

```
1 (defn the-lambda []
2   (println aenv)
3   (println env))
4
5 (defn lambda-wrapper []
6   (the-lambda))
```

closure




```
1  (defn quux []
2    (println aenv)
3    (println env))
4
5  (defn closure []
6    (let [env 12 aenv 17] (quux)))
```

closure implementation

```
1  let codegen_splice_env llenv fname args body =
2    Hashtbl.clear bound_names;
3    Array.iter (fun n -> Hashtbl.add bound_names n) args;
4    let env_vars = extractl_env_vars body in
5    List.iteri (fun i n ->
6                 let elptr = build_in_bounds_gep
7                                 llenv (idx i) "ptr" in
8                 let el = build_load elptr "el" builder in
9                 Hashtbl.add named_values n el;
10               ) env_vars;
11   Hashtbl.add function_envs fname env_vars
```

the final value_t

```
1   %value_t = type {
2        i32,                                     ; type of data
3        i64,                                     ; integer
4        double,                                  ; double
5        i1,                                      ; bool
6        i8,                                      ; char
7        i8*,                                     ; string
8        %value_t**,                              ; array/fenv
9        i64,                                     ; array length
10       %value_t* (i32, %value_t**, ...)*,       ; function
11  }
```

end notes

1. https://github.com/artagnon/rhine
2. http://llvm.org/docs/LangRef.html
3. clang -S -emit-llvm
4. #llvm on Freenode IRC