# Advanced Compiler Techniques

## Advanced projet report

François Bochatay & Yann Randin

## Inlining

### Implementation

Inlining has been implemented as an additional (early) step of the compiler. Inlining takes place just after the name analysis, and thus before closure conversion for example. Inlining has been implemented as a visitor in the Inliner object. The Inliner take the current tree and produces an inlined version of it.

#### Analysis

Here, the analysis must only identify the function that should be inlined and keep their respective body. We decide to inline all top level functions, except the recursive ones. The analysis is thus very simple: parse the tree and at each function definition test if its body contains a recursive call; if not, add its body to the *inlineableFunctions* HashMap. This HashMap is indexed by the symbol of the function name.

#### Modification of the tree

The inline per se is relatively simple. Whenever a function call is encountered and this function is in *inlineableFunctions*, then beta-reduce the function using the arguments. The beta-reduction must also be done when a `lambda` is immediately called. This is done by the two *inline* methods

The beta reduction is done in two phases. The two *beta-reduce* methods act as a frontend, handling general beta-reduction calls. Then, the *do_beta_reduction* actually perform the tree modification as long as the necessary alpha-renaming.

The *beta-reduce* methods handle the treatment of the function parameters: if the parameter is a simple identifier or number, it can simply be replaced in the body; if the parameter is an expression, it must be defined in a `let` which will enclose the body of the inlined function.

The *do_beta_reduction* methods have two goals: replace the parameters by their values and alpha-rename the symbols created in the body of the function being inlined. The values of the parameters are passed as arguments. Whenever a `let` or a `lambda` is encountered, new symbols must be created firstly to avoid conflicts and secondly to ensure symbol unicity when inlining several times a function. Then, the only remaining thing to do is to replace an identifier by its value if it's a parameter or by a new symbol if needed.

### Problems encountered

We encountered three main problems while implementing inlining. First, we hadn't created new symbols for `lambda` parameters in *do_beta_reduction* before figuring out that this created duplicated symbols on some cases. This occurs when a function defining a `lambda` is inlined several times. Then, we lost some time due to the inversion of the parameters identifiers while creating the new symbols which obviously leaded to wrong results at run time.

We then have a problem with the values of the parameters passed to the body of the function to be inline. If a `let` is necessary when the body has several statements, some expressions simply crashed when directly inlined in the function body. This was particulary the case when inlining `cons` in the `%scale-bignum` method of *bignums*. To address this, we decided to define each expression passed as parameter in an enclosing let. This is done in the *beta-reduce* method.

When checking the validity of the inlining, we discovered that immediate lambda calls where not handle correctly when inlining was turned off. So we have to change the way the environment was passed in the closure converter to address this specific case.

## Precise copying garbage collector

### Tagging scheme

#### In the compiler

In order to make the pointer distinguishable from integers, we have decided to use the tagging scheme 2n+1 for every integer present in memory. The changes to the compiler have all been done in the file Generator.scala. This solution prevents further optimizations as it directly produces the compiled code.

This implies:
- All integers n loaded with LINT are in the form 2n+1.
- All arithmetic and logical primitives have been updated to take into account the modification of their operands.
- The space allocation parts (global variable space, stack frame and local variables space of functions) also use the tagging scheme.

Modifications performed to the primitives:

| | |
|---|---|
| case "$+": | 2(n+m) + 1 = (2n+1 + 2m+1) - 1 |
| case "$-": | 2(n-m) + 1 = (2n+1 - 2m+1) + 1 |
| case "$*": | 2(n*m) + 1 = (2n * 2m)/2 + 1 |
| case "$/": | 2(n/m) + 1 = 2(2n / 2m) + 1 |
| case "$%": | as x%y = x - x*(x/y) we have: 2x % 2y = 2x - 2x*(2x/2y) = 2(x - x*(x/y)) = 2(x%y) As the logical primitives work with 0 and 1, which are integers, the result of the comparison has to be tagged: |
| case "$=": | result is stored as 2(x=y) + 1 |
| case "$<": | result is stored as 2(x<y) + 1 |
| case "$<=": | result is stored as 2(x<=y) + 1 |

**In the virtual machine**

- All I/O instructions have been modified so that they add the tagging scheme to incoming integers and characters and remove the tagging before the output.
- The ALOC instruction now has to remove the tagging scheme before performing the memory allocation.

We have define two macro in memory.h in order to factories the tagging modifications in the vm:

```
#define REMOVE_INT_TAGGING(num)            \
   ((num)-1)/2
#define PUT_INT_TAGGING(num)               \
   ((num)*2)+1
```

The macro IS_HEAP_POINTER of the Mark & Sweep GC has been adapted to take advantage of the tagging scheme. By doing so we allow the Mark & Sweep GC to perform better with the tagged "binaries".

```
#define IS_HEAP_POINTER(addr, mem_start, mem_end)          \
(((addr) << 31 == 0) && ((mem_start) <= (char*)(addr)) &&
((char*)(addr) < (mem_end)))
```

## Copying Garbage Collector Implementation

When an allocation request comes, we allocate memory only if there is enough place to hold the new free header in the resulting heap space. This removes the need of a special case for the allocation of the last block of the heap memory.

We have implemented the Cheney's copying GC algorithm. First of all we copy the blocks referenced by pointers in registers. Then we scan the blocks moved to the to-space in order to find pointers to block that have to be copied to the to-space. This ends when all blocks have been scanned i.e. the scan pointer reaches the free pointer.

In order to store the forwarding pointer we used an additional field in the block header named moved_to. This field exist in all block in the heap, even in the free one.

As the scanning of the memory is not aware of the headers, the size field should not be identified as a pointer. This is avoided by the use of the tagging scheme.

When the garbage collecting is executed before a block has been fully filled by the program, we will scan old memory elements which potentially contain no more valid pointers. To address this problem, we clear the old memory space at the end of the garbage collecting.

## Performances

We have run some tests to measure the improvement of the copying GC over the mark & sweep GC.

The tests have been run on a PowerBook G4 1.67Ghz 1Go Ram. All tests were made using the minimum amount of heap necessary to end and the log level set to 2.

Bignums for 100:
- Copying: 0.325s with a heap of 17000
- M&S: 0.636s with a heap of 27000

Bignums for 200:
- Copying: 0.813s with a heap of 32000
- M&S: 3.653s with a heap of 55000

Bignums for 300:
- Copying: 1.707s with a heap of 46000
- M&S: 15.829s with a heap of 85000