

Comp 511 Notes

Inscribed by Anthony Castanares, Seth Fogarty, and Kristin Y. Rozier

February 23, 2005

1 Cyclone

| MetaOCaml | 'C | Cyclone | CIR |
|---------------------|------------------|----------------|-----------------------|
| <> | ' | <i>splice</i> | <i>copy</i> |
| ~ | @ | <i>cut</i> | <i>copy</i> |
| ! | <i>compile()</i> | ?? | |
| ~ (<i>lift!_</i>) | \$ | <i>fill</i> | <i>copy</i> |
| ! < _ > | | <i>codegen</i> | <i>start/end/copy</i> |

Note that there are two versions of this paper. The one we were emailed a link to, with 32 pages, and the one linked of the 511 page with 16 or so pages.

Cyclone has four primary constructs. These constructs have no real analogue in 'C, something that should merit notice. What is going on here that the constructs are so different? On first glance, it is easy to note there are no *lvalue* or *rvalue* distinctions. Indeed, Cyclone constructs operate only over statements (*splice*, *codegen*, *cut*) or rvalues (*fill*). You cannot create arbitrary expressions, only assemble statements and lift values.

It is interesting, however, that both have a *fill* construct. This is evidently important enough that even in two different interpretations of multi-stage programming it shows up.

CIR is an intermediate representation, such as would be used for several languages. This makes the first step in compiling, compiling to the CIR, the only necessary step to add a new language. You can reuse the same back-end and middle-end of your compiler, and you can do type safety and security checks on the simpler CIR.

When reading through the paper, it is important to note that *emit* is a translation construct used in describing compilers. When you see a certain piece of code you emit various instructions in a certain order. Emit is like print, a side effect of translation. Curiously, *emit* and *splice* seem to be very similar.

2 Compiling the RTCG constructs

The source language has *codegen*, *splice*, *cut*, and *fill*. The target language has *start*, *copy*, *fill*, and *end*.

There's an awful lot of renaming that goes on with no explanation, but we are going to ignore this for the time being. None the less it points to a significant difficulty that should probably have been addressed.

The paper talks about *templates*. *templates* are created when compiling. This allows optimizations on the *templates*.

A piece of code which is being assembled and will eventually be compiled at runtime is called a *region*.

You start a region with the *start* command, which is the first thing a translated *codegen* will do.

You *copy templates* into *regions*, which creates *instances*. An instance is a copy of a *template* that needs to be *filled* if there are any holes. Note that holes are created by the presence of a *fill* command, so it is impossible to have a hole that is not *filled*.

So regions look like:

```
start (name) allocates a region
instance
instance
instance
instance
instance
end compiles a region
```

The problem with templates with holes is that holes are not of known length. This can put off your jump targets, and is something that needs addressed. This is a large section of the paper which is concerned with inter-template jumps, which we never discussed in class

3 The Three Copies

There are three places in the code translation in which copies occur: `splice`, `cut`, and `codegen`. The `codegen` translation contains a direct `copy` while the `cut` and `splice` translations contain calls to `newTemplate`, which contains a `copy` instruction. Why are there three copies?

A `codegen` command is followed by a template and a sequence of `cuts`, possibly with templates in between them. The one `copy` in the `codegen` just copies the first template. Therefore, at the end of the work being done by a nested `cut`, another `copy` command is necessary to resume copying again for the rest of the `codegen` region. Referring to the translation on page 18, `cut` has a `copy` statement at the very end while `splice` just falls through. Basically, what `cut` has to do is do the work of splicing in some expression. Then when it is finished, it has to go return to the `codegen` region, copying the next thing that `codegen` would have needed to copy. Conversely, `splice` performs a `copy` at the beginning of its execution and then just falls through to the calling region when it is finished. Nesting these operations allows Cyclone to have multiple levels, like MetaOCaml. This is in contrast to ‘C, which is otherwise more closely related to Cyclone but has the implicit escapes which limit it to two levels.

4 Dissecting codegen

The definition of `codegen` is on page 17. The translation basically consists of a `start`, a `copy`, a translation of f with ϕ , and an `end`. The most interesting thing to note here is that f is being translated with the regular transformation function. Templates contain run-time-generated code which is just not executable. Templates are just blocks of machine code that might contain holes. The authors manage to use the same compilation function for templates as for normal code, contributing to their optimization.

With respect to the notion of parent and child levels, a child level is entered whenever work has begun inside a template. The parent level is entered whenever work has begun inside the active code-generating function. For example, once a `codegen` has begun, the child level has begun and we escape to the parent level at a `cut`. Finally, we go again to the child level when a `splice` is encountered. Remember, there are only two levels in this language, no more.

This concept is clarified when presented in the notion of buckets. When we begin a `codgen`, a bucket is filled with template code and a `copy` statement is appended to the end of it. A `cut` statement brings us back to the parent level, wherein we stitch together all the code-carrying buckets. A `splice` creates another bucket with template code and a `copy` as well.

All the code for the template is generated at the parent level. When a `cut` is entered, there are three operations that take place: First, the argument to `cut` is translated. Second, the code that is produced from the translation is emitted back to the parent level. Finally, a call to `newTemplate` prepares the next evaluation context for the next template.

After a call to `cut`, `newTemplate` does the following: First the name of the active function is acquired. Second, a new variable is initialized with a fresh name using the `newVar` construct. Third, the upper and lower bounds of a new block are initialized using the `changeBlock` construct. Fourth, the name of the template is applied to the current template using `setTemplateOfCurrent`. Finally, the `copy` operation is used to emit code back to the parent level containing the new template generated by the active function.

The `splice` construct always appears within a `cut`. It goes to the child level and translates its argument, and puts the result of that translation into the child bucket. Finally, it puts a `jump` fallthrough in the child bucket, so that evaluation can continue uninterrupted in the parent level.

When comparing the way dynamic code generation works in MetaOCaml, 'C, and Cyclone with templates, it is interesting to note that in Cyclone, one can never get a handle on generated code. It can not be bound to a variable nor can it be passed to another function. In 'C, and in MetaOCaml, it can.