BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

The Beast programming language

Daniel Čejchan*

Abstract
This paper introduces a new compiled, imperative, object-oriented, C-family programming language, particularly inspired by C++ and D. Most notably, the language implements a new concept called *code hatching* that unites templating, compile-time function execution, reflection and generally metaprogramming. The project also includes a proof-of-concept compiler (more precisely transcompiler to C) called Dragon that demonstrates core elements of the language (downloadable from the Git repository).

Keywords: Programming language — CTFE — code hatching — compile time — metaprogramming — ctime — Beast

Supplementary Material: Git repository (github.com/beast-lang/beast-dragon)

*xcejch00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

There are two ways of how to approach introducing Beast – either it can be referred as a programming language designed to provide a better alternative for C++ programmers or as a programming language that implements the code catching concept, which introduces vast metaprogramming and compile-time computing possibilities.

As a C++ alternative, the language provides syntax and functionality similar to C++, but adds features designed to increase coding comfort, code readability and safety. The most notable changes are:

- Instead of header/source files, Beast has modules with importing system similar to D or Java.
- Beast variables are const-by-default. As C++ had the **`const`** keyword to mark variables constant, Beast has the `Type!` suffix operator to make variables not constant.
- References and pointers are designed differently. In Beast, references are rebindable and can be used in most cases where C++ pointer would be used. Only when pointer arithmetic is needed, Beast pointers have to be used.

There are many smaller changes; those are (or will be) documented in the language reference and bachelor thesis text (both downloadable from the Git repository).

As mentioned before, another aspect of Beast is its *code hatching* concept, which is based on one simple thought – having a classifier for variables whose value is deducible during compile time. In Beast, those variables are classified using the `@ctime` decorator – for example `@ctime Int x;` They can be mutable (using the `!` operator, like with standard variables; this applies only to local variables – because Beast code is not processed linearly, order of evaluation of expressions modifying static `@ctime` variables could not be decided) and they even can be mixed with standard code (although their mutation can never depend on non-`@ctime` variables or inputs).

With this establishment, it is possible to start con-

```
@ctime Int z = 5;

Void main() {
  @ctime Int! x = 8;
  Int! y = 16;
  y += x + z;
  x += 3;
}
```

Figure 1. Example of mixing `@ctime` and non-`@ctime` variables in Beast (`@ctime` static variables cannot be mutable)

sidering class and generally type definitions as `@ctime` constant variables (thus first-class citizens).

```
Void main() {
  @ctime Type! T = Int;
  T x = 5;
  T = Bool;
  T b = false;
}
```

Figure 2. Example of using type variables in Beast

Having type variables, templates can now be considered functions with `@ctime` parameters. Class templates become functions returning a type. With `@ctime` variables, generics, instead of being a standalone concept, become a natural part of the language.

```
auto readFromStream( @ctime Type T,
   Stream!? stream )
{
  T result;
  stream.readData( result.#addr,
     result.#sizeof );
  return result;
}

Void main() {
  Int x = readFromStream( Int, stream );
}
```

Figure 3. Example of function with `@ctime` parameters in Beast

Adding compile-time reflection is just a matter of adding compiler-defined functions returning appropriate `@ctime` data.

The `@ctime` decorator can also be used on more syntactical constructs than just variable definitions:

- `@ctime` code blocks are entirely performed at compile time.
- `@ctime` branching statements (**`if`**, **`while`**, **`for`**, etc.) are performed at compile time (not their bodies, just branch unwrapping).
- `@ctime` functions are always executed at compile time and all their parameters are `@ctime`.
- `@ctime` classes can only be constructed at compile time (for instance, `Type` is a `@ctime` class)

There is also a set of rules that apply on `@ctime` variables, functions and such. Those rules are deduced in author's bachelor thesis [1] (downloadable from the Github repository).

1. `@ctime` variables cannot be data-dependent on non-`@ctime` variables.
2. Static `@ctime` variables must not be mutable.
3. If a variable is `@ctime`, all its member variables (as class members) are also `@ctime`.
4. If a reference/pointer is `@ctime`, the referenced data is also `@ctime`.
5. `@ctime` variables can only be accessed as constants in a runtime code.
6. `@ctime` references/pointers are cast to pointers/references to constant data when accessed from runtime code.
7. A non-`@ctime` class cannot contain member `@ctime` variables.

```
Void main() {
  @ctime if( true )
    println( "Yay!" );
  else
    println( "Nay! );

  @ctime for( Int! x = 0; x < 3; x ++ )
    print( x );
}

// Is processed into:
Void main() {
  println( "Yay!" );
  print( 0 );
  print( 1 );
  print( 2 );
}
```

Figure 4. Example of `@ctime` branch statements in Beast

2. Implementation details

It is clear that an implementation of the code hatching concept requires the compiler to be able to execute any code during compilation. However for optimal performance, two different interpreters are needed.

In Figure 6, it is not possible to compile the code into standard assembly (or intermediate code) until variable `a` and `b` declarations (which depend on `@ctime` variable `T`) are resolved. This means that the `@ctime` code has to be executed in-place and the semantic tree must be built simultaneously with the `@ctime` code

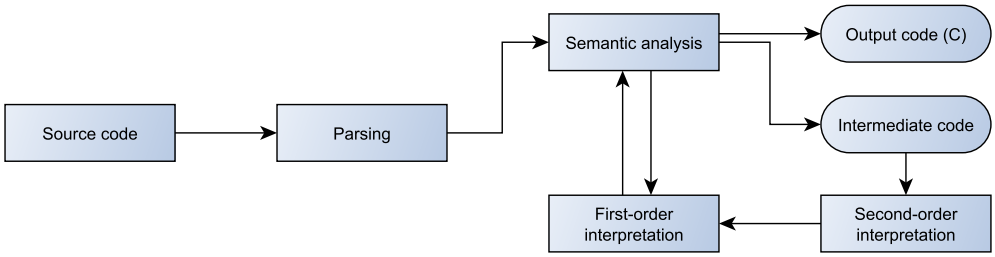

Figure 5. Diagram of code processing workflow in Dragon (Beast compiler)

```
Void main() {
  for( Int! x = 0; x < 50; x ++ ) {
    @ctime Type! T = Int;
    T a = 5;
    T = Bool;
    T b = true;
  }
}

// Is processed into:
Void main() {
  for( Int! x = 0; x < 50; x ++ ) {
    Int a = 5;
    Bool b = true;
  }
}
```

Figure 6. Demonstration of two-level interpretation

being interpreted (also called *first-order interpretation*). Results of the *first-order interpretation* directly affect further variable definitions and call bindings (as demonstrated in Figure 7), so intermediate code for a first-order interpreter cannot contain more information than an abstract syntax tree (AST). That means that intermediate code generation for the first-order interpreter is pointless.

```
Void main() {
  @ctime Type! T = ClassA;
  @ctime if( 5 > 6 )
    T = ClassB;

  @ctime T var; // Variable type
     depends on previous @ctime
     interpretation
  var.func(); // What function is
     called (even though being
     statically bound) depends on
     previous @ctime interpretation
  )
}
```

Figure 7. Example where `@ctime` variables cannot be optimized out

Although it is possible to use the first-order interpreter to interpret both `@ctime` and non-`@ctime` code during compilation, building an intermediate code is more suitable if a piece of code is executed more than once - this applies to code inside loops or any function that is executed more than once. As explained before, it is pointless to build an intermediate code for the first-order interpreter, but it can be done after the `@ctime` code is processed. The interpreter that processes this intermediate code that is generated after the semantic analysis and first-order interpretation is called *second-order interpreter*.

Contrary to intuition, during the first-order interpretation, `@ctime` variables cannot always be fully optimized out from the output code.

```
Void println( String? str ) {
  // ...
}

Void main() {
  @ctime String asdStr = "asd";
  println( asdStr );
  asdStr = "notAsd";
  println( asdStr );
}
```

Figure 8. Example where `@ctime` variables cannot be optimized out

In Figure 8, we are passing a `@ctime` variable `asdStr` to a runtime function `println` via reference. The function cannot be executed at compile time, because it prints into standard output, and it does not know that the string we are passing to it is `@ctime`. This means that the variable `asdStr` must actually have an address assigned at runtime and that address must contain valid data. When the value of the `@ctime` variable is changed, those changes have to be mirrored in the runtime memory – an appropriate set of instructions in the target code has to be generated to update the value of the `@ctime` variable.

There is still a big difference between `@ctime` and non-`@ctime` variables. First, this "runtime mirroring" is not necessary at all in a lot of practical use cases. Second, the mirroring can be a lot faster than runtime execution – mirroring does not require the evaluation to be done at runtime, a simple memory overwrite with newer values (and only where the `@ctime` data changed

since the last mirroring) is enough.

The compiler has a virtual memory address space that is used during interpretation. This address space is different from the address space of an output binary file. Since it is possible to have references (and pointers) during compile time, it is necessary to implement address translation between these two address spaces. Because of that, the compiler has to know what data in the virtual memory is a reference, because values of `@ctime` references are changed during the linking process. This is realized by dedicated interpreter instructions called within reference (pointer) constructor/destructors that mark/unmark the memory as a reference.

This text briefly describes all key components necessary to implement the code hatching concept. More in-depth analysis can be found in bachelor thesis downloadable from the Git repository.

3. Existing solutions

Beast is inspired by the D Programming Language [2] that also has vast metaprogramming and compile-time execution capabilities. However in D, compile-time constants cannot be mutable. Although code can be executed at compile time, there are no type variables, so working with types usually ends up in definition of recursive templates.

Among imperative compiled languages, there are no other well established programming languages with such metaprogramming capabilities. However, recently several new programming language projects introducing compile-time capabilities emerged – for example Nim [3], Crystal [4], Ante [5] or Zig [6].

Acknowledgements

I would like to thank my supervisor, Zbyněk Křivka, for his supervision, and Stefan Koch (on the internet known as UplinkCoder) for the time he spent chatting with me about this project.

References

[1] Daniel Čejchan. Compiler for a new modular programming language, 2017.

[2] D programming language, c1999-2017.

[3] Nim programming language, c2015.

[4] The crystal programming language, c2015.

[5] Ante: The compile-time language, 2015.

[6] The zig programming language, 2015.


```
class C {

@public:
  Int! x; // Int! == mutable Int

@public:
  // Operator overloading,
     constant-value parameters
  Int #opBinary(
    Operator.binaryPlus,
    Int other
    )
  {
    return x + other;
  }

}

enum Enum {
  a, b, c;

  // Enum member functions
  Enum invertedValue() {
    return c - this;
  }
}

String foo( Enum e, @ctime Type T ) {
  // T is a 'template' parameter
  // 'template' and normal parameters
     are in the same parentheses
  return e.to( String ) + T.#identifier;
}

Void main() {
  @ctime Type T! = Int; // Type
     variables!
  T x = 3;

  T = C;
  T!? c := new C(); // C!? - reference
     to a mutable object, := reference
     assignment operator
  c.x = 5;

  // Compile-time function execution,
     :XXX accessor that looks in
     parameter type
  @ctime String s = foo( :a, Int );
  stdout.writeln( s );

  stdout.writeln( c + x ); // Writes 8
  stdout.writeln(
     c.#opBinary.#parameters[1].type.#identifier
     ); // Compile-time reflection
}
```

Figure 9. Beast features showcase (currently uncompilable by the proof-of-concept compiler)