BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

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

The Beast programming language

Daniel Čejchan*

Beast
PROGRAMMING LANGUAGE

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* (also a subject of this paper) that unifies templating, compile-time function execution, compile-time reflection and metaprogramming in general. The project also includes a proof-of-concept open-source compiler (more precisely transcompiler to C) called Dragon that demonstrates core elements of the language and the code hatching concept.

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 hatching 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++ has the **`const`** keyword to mark variables constant, Beast has `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++ pointers would be used. Beast pointers are to be used only when pointer arithmetic is needed (or double indirection, as there cannot be references of references).
- The `#` character is a valid identifier character. It is used as a prefix for reflection or compiler-related properties and functions such as `variable.#type`, `Type.#instanceSize` or `var.#explicitCast( TargetType )`.

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).

The main innovation of Beast is its *code hatching* concept that unifies formerly standalone concepts of templates, compile-time function execution (CTFE), compile-time reflection and metaprogramming in general (conditional compilation, etc.). The concept blurs borders between standard and templated functions, between code and *metacode* (in C++, an example of metacode would be preprocessor directives or template declarations).

The D programming language, which Beast is inspired by the most, offers all of the functionality mentioned above, however the concepts are implemented rather in the standalone way. Beast brings improvement to the following aspects:

1. D has a dedicated template argument list similar to C++ (the syntax is `!(args)` instead of `<args>` and the `!` is omitted in declarations), making compile-time parameters clearly separated from runtime ones. Functions that differ only in one parameter being compile-time (template) or not have an extensively different syntax (as shown in Figure 1).
   Beast has one common parameter list for runtime and compile-time parameters, resulting in zero syntax difference between runtime and compile-time parameters. It is even possible to use parameters in a single function declaration to work both as runtime or compile-time depending on the context – if the provided argument can be evaluated at compile time, it is considered a template parameter, otherwise it is considered to be runtime (demonstrated in Figure 2).
2. The D programming language does not have mutable compile-time variables, which makes solving some problems impossible with iteration, forcing programmers to use recursion (or mixins), which often results in a hardly-readable code (demonstrated in Figure 3). Beast supports mutable compile-time variables (see Figure 4).

```
// D code
string format( Args ... )( string fmt,
   Args args ) { ... }
string format( string fmt, Args ... )(
   Args args ) { ... }

void main() {
  auto rt = format( "%s, %i worlds",
     "hello", 5 );

  auto ct = format!( "%s, %i worlds" )(
     "hello", 5 );
}
```

Figure 1. Usage of the `format` function in the D programming language (similar to `sprintf` in C). Second `format` definition and function call accept the `fmt` string as a template parameter, resulting in the string formatting code being generated at compile time.

```
1 String format( @autoctime string fmt, auto
     arg ... ) { ... }
2
3 Void main() {
4   auto rt = format( Console.readln, "hello",
       5 );
5
6   auto ct = format( "%s, %i worlds", "hello",
       5 );
7 }
```

Figure 2. Beast code corresponding to D code in Figure 1. On line 4, the `fmt` argument cannot be evaluated at compile time, resulting in it being considered a runtime parameter. On line 6, the argument can be evaluated at compile time, resulting in it being treated as `@ctime` (compile-time, template) and in string formatting code being generated at compile time.

```
// D code
template memberTypes1( Type ) {
  alias memberTypes1 = helper!( __traits(
     allMembers, Type ) );

  template helper( string[] members ) {
    static if( members.length )
      alias helper = TypeTuple!(
        typeof( _traits( getMember, Type,
           members[ 0 ] ) ),
        helper!( member[ 1 .. $ ]
        );
    else
      alias helper = TypeTuple!();
  }
}

template memberTypes2( Type ) {
  mixin( {
    string[] result;

    foreach( memberName; __traits(
       allMembers, Type ) )
      result ~=
        "typeof( __traits( getMember, Type,
           %s ) )"
        .format( memberName );

    return "TypeTuple!( %s )".format(
       result.joiner( ", " ) );
  }() );
}
```

Figure 3. Two approaches of writing a 'function' returning a `TypeTuple` (compile-time analogy to an array) of types of members of given type `Type` in the D programming language. First approach uses recursion, second one mixins.

```
@ctime Type[] memberTypes( Type T ) {
  Type[]! result;

  foreach( auto member; T.#members )
  result ~= member.#type;

  return result;
}
```

Figure 4. Beast function corresponding to D 'functions' from Figure 3. The function returns array of types of members of given type `T`.

2. Principles of code hatching

The code hatching concept is based on a simple idea – having a classifier for variables whose value is deducible during compile time. In Beast, those variables are classified using the `@ctime` decorator (usage `@ctime Int x;`). Local `@ctime` variables can be mutable; because Beast declarations are not processed in order as they are declared in source code, order of evaluation of expressions modifying static `@ctime` variables cannot be decided; that means that static `@ctime` variables cannot be mutable. All `@ctime` variable manipulations are evaluated at compile time.

`@ctime` variables can also be included within a standard code (although their mutation can never depend on non-`@ctime` variables or inputs).

```
@ctime Int z = 5;

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

Figure 5. Example of mixing `@ctime` and non-`@ctime` variables in Beast

The concept of variables completely evaluable at compile time brings a possibility of having type variables (only `@ctime`, runtime type variables cannot be effectively done in compiled, statically typed languages). As a consequence, class and type definitions in general can be considered `@ctime` constant variables (thus first-class citizens).

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

Figure 6. Example of using type variables in Beast

Having type variables, templates can be viewed as functions with `@ctime` parameters, for example class templates can be viewed 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 7. Example of a 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` expressions are always evaluated at compile time.
- `@ctime` class instances can only exist as `@ctime` variables (for instance, `Type` is a `@ctime` class)

To make the code hatching concept work, it is necessary to ensure that `@ctime` variables are truly evaluable at compile time. That is realized by the following rules. Their deduction can be found in author's bachelor thesis [1] (downloadable from the Github repository).

1. `@ctime` variables cannot be data-dependent on non-`@ctime` variables.
   (a) Data of non-`@ctime` variables cannot be assigned into `@ctime` variables.
   (b) It is not possible to change `@ctime` variables declared in a different runtime scope; for example it is not possible to change `@ctime` variables from a non-`@ctime` if body if they were declared outside it.
2. Static (non-local) `@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( "Hello, %s!".format( "world" ) );
  else
    println( "Nay! );

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

  print( @ctime "Goodbye, %s!".format( "world" ) );
}

// Is processed into:
Void main() {
  println( "Hello, %s!".format( "world" ) );
  print( 0 );
  print( 1 );
  print( 2 );
  print( "Goodbye, world!" );
}
```

Figure 8. Example usage of the `@ctime` decorator in Beast

3. Existing solutions

Beast is inspired by the D Programming Language [2]. Differences between Beast and D are described in Section 1.

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]. From the list, Zig is the most similar language to Beast. Author of Beast and the code hatching concept was not aware of existence of Zig during the language design, so the two languages emerged independently.

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.binPlus,
    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 auto(); // 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)