CREATING THE BOLT COMPILER: PART 10

Generics - adding polymorphism to Bolt

JANUARY 23, 2021 4 MIN READ

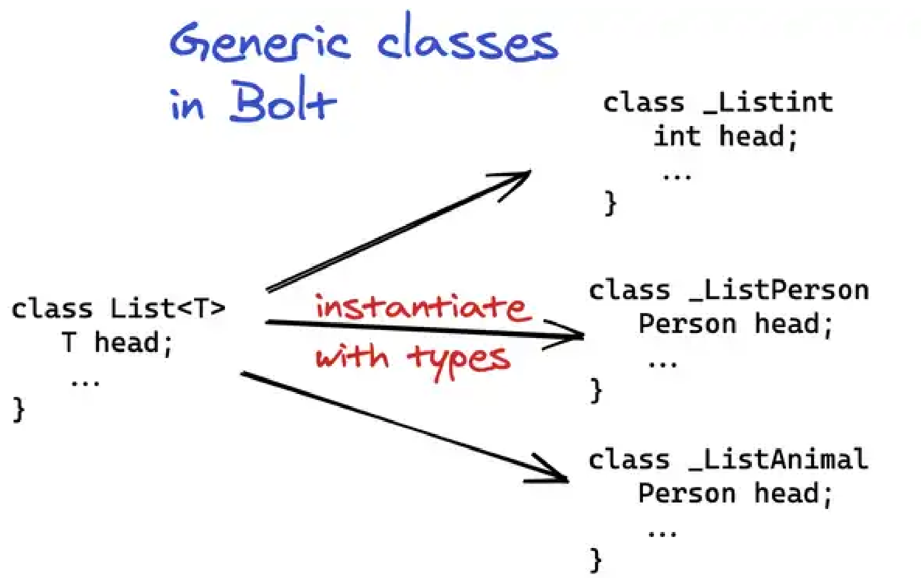

TABLE OF CONTENTS

SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt

Part 11: Adding Inheritance and Method Overriding to Our Language

---

Onward with more features that any "proper" programming language needs. Today we're implementing **generics**. Generics allow you to reuse code for multiple types. Take a `List` for example. A list has the same operations regardless of types: it'd be a pain to write out a new class for each list.

Copy

```
class ListInt{
  ...
}
class ListPerson{
  ...
}
class ListAnimal{
  ...
}
```

The generic class for that would be `List<T>`. We call `T` the generic type *parameter*. Think of it like a variable, which we assign a type to when we instantiate the class: `List<int>()`. Let's build it! We'd like to compile this program:

Copy

```
class List<T>{
  void add(T a){
    ...
  }
  T getHead(){
    ...
  }
  int size(){
    ...
  }
}

void main(){
  let list1 = new List<int>();
  list1.add(4);
  ...
```

```
}
```

Just give me the code!

As ever, the code is in the Bolt repo. The generics are handled in the typing and desugaring stages of the compiler. The code is in the files that contain `generics` in their name e.g. `type_generics.ml`. You could even just search "generic" in the repo!

Type parameters are just like other types!

Rejoice, we don't need to rewrite our type-checker! This tutorial is much shorter than you think. Here's all that changes in the type-checker:

- **Within** our generic class, we can treate our type parameter `T` as an opaque type `TEGeneric`. So type-check the class as before, just don't make any assumptions about `T`.
- Outside a generic class we can't use generic types `T`, so raise an error if we see that.
- Whenever we use an object instantiated with a type, e.g. `List<int>`, we can replace all occurrences of `T` with the instantiated type `int`!

That's all that's changed. Seriously!

Treat the generic type parameter as an opaque type

We've added to the list of Bolt types a `TEGeneric` type to represent this opaque type `T`.

When we call objects of generic classes, we don't have an object of just `List`, it's `List<int>`, `List<Person>` etc. So we update class types `TEClass` to carry around this instantiated type parameter `int`, `Person` etc. if they're generic.

`ast_types.ml`

Copy

```
type type_expr =
```

```
  | TEInt
  | TEClass   of Class_name.t * type_expr option  (** optionall
  | TEVoid
  | TEBool
  | TEGeneric
```

Next we need to update the `class_defn` type to distinguish between non-generic and generic classes. We define a special type as I think it's more instructive to see `Some Generic | None` rather than `true | false`. As before, ignore the `capability list` if you're not interested in the data-race prevention! (See my dissertation for a full explanation if you are).

ast_types.ml

Copy

```
type generic_type = Generic
```

parsed_ast.ml

Copy

```
type class_defn = TClass of
      Class_name.t
      * generic_type option
      * capability list (* for data-races (see dissertation) *)
      * field_defn list
      * method_defn list
```

And now, within a generic class `List`, we instantiate `this` to be of type `List<T>` (remember we treat `T` as an opaque type `TEGeneric`):

type_generics.ml

Copy

```
let instantiate_maybe_in_generic_class_this
    (Parsed_ast.TClass (class_name, maybe_in_generic_class, _,
  let maybe_type_param =
    (* use generic type T inside class *)
    match maybe_in_generic_class with Some Generic -> Some TEGe
```

```
    (Var_name.of_string "this", TEClass (class_name, maybe_type_p
```

And then the type-checking works as before!

Check usage of generic types

Outside a generic class we can't use generic types. I hate to bore you, this code is quite mechanical - it's a lot of recursively going through each of the subexpressions. For a class, check each of the fields, methods etc. For a function, check its type signature and then its body. And so on.

Here's a snippet of a function that checks a type. If we're in a generic class it's all fine, otherwise check we aren't using a generic type. Click the link to the `type_generics.ml` file below to see the full code.

[type_generics.ml]

Copy

```
let rec type_generics_usage_type type_expr maybe_in_generic_cla
  match maybe_generic with
  | Some Generic -> Ok () (* can have generics in generic class
  | None         -> (
    (* recursively check there aren't nested uninitialised type
    match type_expr with
    | TEInt | TEBool | TEVoid       -> Ok ()
    | TEGeneric                     ->
      Error
        (Error.of_string
           (Fmt.str "%s Type error: Use of generic type but n
              error_prefix_str))
    | TEClass (_, maybe_type_param) -> (
      match maybe_type_param with
      | Some type_param ->
          type_generics_usage_type type_param maybe_in_generic_
      | None            -> Ok () ) )
```

Instantiate generic objects

We check first that we should be instantiating with a type-parameter. If we're trying to instantiate a non-generic class with a type param, raise an Error, and likewise if we haven't provided a concrete type for a generic class, raise an error. If we do have a generic class, then recursively replace all instances of a generic type with the concrete type: the fields and then the methods etc. Again, full details are in the repo:

[type_generics.ml]

Copy

```
let instantiate_maybe_generic_class_defn maybe_type_param
    ( Parsed_ast.TClass
        (class_name, maybe_generic, caps, field_defns, method_d
    class_defn ) loc =
  match (maybe_generic, maybe_type_param) with
  | None, None  (* non-generic class *) ->  Ok class_defn
  | None, Some type_param         ->   Error ...
  | Some Generic, None            -> Error ...
  | Some Generic, Some type_param ->
      List.map ~f:(instantiate_maybe_generic_field_defn type_pa
      |> fun instantiated_field_defns ->
      List.map ~f:(instantiate_maybe_generic_method_defn type_p
      |> fun instantiated_method_defns ->
      Ok
        (Parsed_ast.TClass
           ( class_name
           , maybe_generic
           , caps
           , instantiated_field_defns
           , instantiated_method_defns ))
```

Desugaring Generics

Ok, so we've type-checked our generics, and they pass our checks. What now? What do we tell our LLVM compiler backend to do when it encounters a `T`? You can't allocate a "generic" block of memory.

So we *desugar* away all mentions of generic types. What the compiler backend doesn't know about, it doesn't have to deal with.

Remember, we did this for function overloading [in our desugaring post]:

Copy

```
function int test(int f) {
  ...
}
function int test(bool b){
  ...
}

// DESUGARED (name-mangle functions)

function int testi(int f) {
  ...
}
function int testb(bool b){
  ...
}
```

The compiler backend doesn't need to worry about multiple functions with the same name, because we handled it in the desugaring stage.

Remember how I said it'd be a pain to write out a new class for each list? It would be *for us*, as we're doing it by hand. It isn't for the compiler: it can automate it! To avoid any name-clashes, we'll prepend each compiler-generated class with an `_`.

Copy

```
class _Listint{
  ...
}
class _ListPerson{
  ...
}
class _ListAnimal{
  ...
}
```

So our desugaring stage has 3 steps to handle generics:

- Count all instantiations of generics

- Create a special class for each of the instantiations (identical to how we instantiated generic objects earlier)
- Replace each generic class' constructor with its instantiated class. So `List<int>` goes to the class `_Listint`.

As before, let's dive into the code!

Count all instantiations of generics

This is in the `count_generics_instantiations.ml` file in the repo (creative name I know!).

We go through the code recursively, and every time we see a constructor with a concrete type param e.g. `List<int>`, we add that instantiation `int` to the total instantiations. In the code below, `class_insts` is a list containing pairs `(class_name, list_of_types_instantiated_with)`:

count_generics_instantiations.ml

Copy

```
let rec count_generics_instantiations_expr class_defns expr cla
  match expr with
  | Typed_ast.Constructor (_, class_name, maybe_type_param, con
      ( match maybe_type_param with
      | Some TEGeneric  -> class_insts (* only consider concret
      | Some type_param -> add_instantiation class_defns type_p
      | None            -> class_insts )
  ... (* recursive calls *)
```

> An aside: we can be overly conservative with our counting, as if we instantiate classes that don't actually get used, then LLVM will optimise them away. So we could have brute-forced all possible combinations - this would have slowed the compiler down, but it wouldn't have affected the code output.

Replace generic classes with instantiated classes

The first step is to replace the class definitions: below we instantiate all the generic classes with concrete types, then filter the original generic classes out and return the updated list of classes.

[replace_generic_with_instantiated_class_defns.ml]

Copy

```
let replace_generic_with_instantiated_class_defns class_defns c
  List.map
    ~f:(fun (class_name, type_params) ->
      List.find_exn
        ~f:(fun (Typed_ast.TClass (name, _, _, _, _, _)) -> nam
        class_defns
      |> fun class_defn -> instantiate_generic_class_defn type_
    class_insts
  |> fun instantiated_class_defns ->
  (* get rid of uninitialised generic classes *)
  List.filter
    ~f:(fun (Typed_ast.TClass (_, maybe_generic, _, _, _, _)) -
      match maybe_generic with Some Generic -> false | None ->
    class_defns
  |> fun non_generic_class_defns ->
  List.concat (non_generic_class_defns :: instantiated_class_de
```

We then need to replace all references to generic classes in the program with the special instances (we name-mangle them). Inside the compiler we convert `List<int>` to `_Listint` . Again, the code is mechanical and a lot of recursive cases replacing generic class names with the new instantiated class:

[name_mangle_generics.ml]

Copy

```
let name_mangle_generic_class class_name type_param =
  Class_name.of_string
    (Fmt.str "_%s%s" (Class_name.to_string class_name) (string_
```

Summary

That's it! We only had to modify our type-checker and desugaring stage to handle generics, and most of the code was just going through each sub-expression recursively.

This approach of replacing a generic class with specialised instances (one for each concrete type) is called **monomorphism** and it is what C++ does with its templates. If you want to find out more about how other languages implement generics, check out this blog post for a more technical read.

Share This On Twitter

If you liked this post, please consider sharing it with your network. If you have any questions, tweet away and I'll answer :) I also tweet when new posts drop!

PS: I also share helpful tips and links as I'm learning - so you get them **well before** they make their way into a post!

Post Follow @mukulrathi_

SERIES: CREATING THE BOLT COMPILER

Part 1: How I wrote my own "proper" programming language
Part 2: So how do you structure a compiler project?
Part 3: Writing a Lexer and Parser using OCamllex and Menhir
Part 4: An accessible introduction to type theory and implementing a type-checker
Part 5: A tutorial on liveness and alias dataflow analysis
Part 6: Desugaring - taking our high-level language and simplifying it!
Part 7: A Protobuf tutorial for OCaml and C++
Part 8: A Complete Guide to LLVM for Programming Language Creators
Part 9: Implementing Concurrency and our Runtime Library
Part 10: Generics - adding polymorphism to Bolt
Part 11: Adding Inheritance and Method Overriding to Our Language

← Write It and They Will (Eventually) Come

Adding Inheritance and Method Overriding to Our Language →