Generic function support for LPython

1. Abstract

Generics are a common functionality found in statically typed programming languages, allowing easier maintenance of programs which differ only in their types. However, generics are not yet supported in the statically-typed LPython. Generics in LPython can be implemented similar to the approach taken by C++ templates. We can regard generic functions as functions with type parameters, whose parameters are made concrete and checked by function calls on AST level.

The end result of the project would be generic function support for LPython with option for specialization, along with sufficient integration tests and documentations.

2. Project background

2.1. LPython

LPython is an ahead-of-time compiled, statically-typed subset of the Python programming language. Programs written in LPython are parsed into ASTs, then compiled into abstract semantic representations, before compiled into LLVM codes.

In its current state, LPython requires all functions to be strongly typed. Thus, unlike Python, we must define an addition function as follows.

```
def add(a: i32, b: i32) -> i32:
  return a + b
```

Where `a: i32` and `b: i32` denote that the parameter `a` and `b` are typed as 32-bit integer and `-> i32` denotes that the return type of function `add` is also a 32-bit integer.

2.2. Generics

Since functions are strongly typed, given different argument and return types, different functions have to be defined in LPython. For example, if we require addition functions for `f32` (32-bit float) and `c32` (32-bit complex), we end up with the following functions.

```
@overload
def add(a: i32, b: i32) -> i32:
  return a + b

@overload
def add(a: f32, b: f32) -> f32:
  return a + b

@overload
def add(a: c32, b:c32) -> c32:
  return a + b
```

This is not user-friendly, as it requires users to write multiple identical functions and at the same time makes maintenance of the program hard. With generics, we can rewrite the functions in Python 3 generics style into:

```
T = TypeVar('T')
def add(a: T, b: T) -> T:
  return a + b
```

Where `T` represents a type parameter that can be parameterized by LPython's native types. Then, when the user provides two different uses of the function, such as `add(1, 1)` and `add(1.0, 1.0)`, the appropriate functions will be determined in compile time.

3. Implementation

3.1. General idea

The generics functionality will be treated as a pass after the program is parsed into an AST and before the AST is converted into an ASR. This corresponds to adding a function that takes an AST and produces a new AST in between calling `parse_python_files` and `python_ast_to_asr` inside the function `compile_python_to_object_file` in `lpython.cpp`. Generics will require a separate module in the compiler.

3.2. Plan

The implementation follows Python 3 syntax for typing, treating type variables similar to variables carrying values. To explain the implementation, I refer back to the `add` function in the Project Background section.

3.2.1. Syntax support

To have generics, we need syntax support for declaring type variables. We need to implement the built-in function `TypeVar`. In the generics pass over the AST, for each occurrence of `TypeVar`, we store the new type variable in a data structure that can be accessed by function definitions. By declaring

```
T = TypeVar('T')
```

Functions underneath it now can declare `T` as the type of their signatures.

3.2.2. Parameter checks on functions

For each function with generic types found during generics pass, we have to implement check on its parameters and return types. With the function definition

```
def add(a: T, b: T) -> T:
  return a + b
```

We check whether `T` is a valid type parameter by referring to the data structure that carries declared type variables.

3.2.3. Type checking function calls

Next would be to implement type check on function calls and instantiation of generic functions' type parameters. For example, given the following function call `add`.

```
add(1,1)
```

We have to check whether the arguments match the function's signature. In this case, `add` requires that both parameters have the same type `T`. `add(1, 1)` passes two identical values `1` (assumed to be `i32`). If we replace `T` with `i32`, then the arguments pass.

Furthermore, if we have a nested function call

```
add(add(1,1), 1) # nested function call
```

We can infer the type of the first argument `add(1,1)` in this process because we can replace `T` with `i32`.

However, the following function call

```
add(1.0, 'a')    # calling with mix typed arguments
```

does not pass the check because we cannot resolve what `T` is.

3.2.4. Tracking function overloadings

For each generic function call that passes the check, we also need to track the substitution for the type parameters. For example, suppose we have two function calls `add` with different arguments

```
# function calls

add(1,1)        # T is replaced by i32
add('a','b')    # T is replaced by str
```

The generics pass assigns to the generic function `add` a different mapping for `T` to both `i32` and `str`.

3.2.5. Generating non-generic functions

The final step is to replace the generic functions in the AST with non-generic functions. For example, since we already have two mapping for `T`, namely `i32` and `str`. We can then generate non-generic `add` functions with these mappings. Using the overloading feature already available in LPython, we can generate the following two functions

```
# resulting AST

@overload
def add(a: i32, b: i32) -> i32:
  return a + b

@overload
def add(a: str, b: str) -> str:
  return a + b

add(1,1)
add('a','b')
```

3.2.6. Support for specialization

We can also add support for specialization. Suppose the implementation for the function is different for each type, we can introduce a decorator `@specialization` for functions implemented for specific types. For

example

```
T = TypeVar('T')

def minus(a: T) -> T:
  return -a

@specialization
def minus(a: bool) -> bool:
  return !a
```

When `minus` is called, the specialized functions are prioritized before attempting to use the generic function.

4. Timeline

June 13 - 19

Work on adding the built-in function `TypeVar` and parsing checks for `TypeVar`.

June 20 - 26

Implement function parameter checks based on declared type variables.

June 27 - July 17

Implementing type checks on generic function calls. The typing visitor goes through function calls and instantiate type parameters according to the arguments given. If unmatching types were found, then the program can return an error.

July 18 - 31

Implement the non-generic functions code generation in AST. A code generating visitor can remove `TypeVar` declarations and generate functions based on how the parameters were instantiated by function calls.

August 1 - 7

Implement specialization support.

August 8 - 21

Testing, fixing bugs, evaluate performance.

August 22- September 28

Further tests, write up documentations.

August 29 - September 4

Write up final report