Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jan Wróblewski
Nr albumu: 277632

Konwersja Pythona do Lukrecji

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr hab. Aleksego Schuberta
Instytut Informatyki

Lipiec 2014

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Abstract

With increasing popularity of dynamic programming languages such as Python or JavaScript the problem of statically typing them has become more important. As Python is a complex language, creating type-checking rules for it requires much more effort than for simpler dynamic languages. Lucretia is such a simple dynamic language for which type checking rules have been defined in a paper by Marcin Benke, Vivania Bono and Aleksy Schubert (see paper [1]). If Python code would be fully conversible to Lucretia code, problem of typing Python could be reduced to typing much less complex language. This work is about a software that converts subset of Python 3 into Lucretia and interprets Lucretia language. It is a part of a larger work with goal of statically typing sufficiently large subset of Python 3.

Supported subset of Python 3 is still not large enough for practical purposes, but most language constructs such as variables, scopes, functions and `while` loops have been implemented. Large part of this work is partial conversion of Python classes into Lucretia. This includes mechanism of class inheritance along with method resolution (with implemented support for single inheritance) and few core classes (such as `object`) that have to be implemented to be able to use classes at all. Conversion was done with type checking in mind, therefore instead of basing implementation on maps from *strings* into values (like in Python), approach to use records with labels known at compile time was taken. That way extensive usage of dependent types in type checker might be avoided.

Lucretia interpreter was completed along with small Lucretia library and mechanisms allowing to write and import Lucretia modules. Few tools helpful in debugging were also implemented in the interpreter, therefore it is possible to create larger Lucretia programs and libraries. This framework was used to create a Lucretia module that simulates part of Python's `builtins` module, needed to use more advanced Python constructs (e.g. classes).

Słowa kluczowe

classes, language conversion, language interpreter, Lucretia, programming languages, Python

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D.3 PROGRAMMING LANGUAGES
D.3.3 Language Constructs and Features
D.3.4 Processors

F.3.2 Semantics of Programming Languages
F.3.3 Studies of Program Constructs

Tytuł pracy w języku angielskim

Conversion of Python to Lucretia

Contents

Chapter 1

Introduction

1.1. Motivation

Lucretia is a dynamic, imperative, functional, object-based, reflective programming language developed by Marcin Benke, Vivania Bono and Aleksy Schubert. Its main purpose is to be a simple dynamic, object-based language that can be statically typed in most real-world scenarios. The type-checking rules and further description of this language are defined in [1]. This paper will focus on translating Python–a popular dynamic language–to Lucretia. The main application of such transformation is to create a Python static type analyzer that would first translate Python to Lucretia and then use Lucretia type-checking rules.

1.2. Lucretia language description

1.2.1. Grammar

For the purpose of this paper Lucretia grammar is defined in Figure 1.1. [X Y] denotes Y-separated list of X. Starting nonterminal of this grammar is Expr, which is a Lucretia program. Terminals are represented using `monospaced font`, either explicitly (like `new`) or using their description in parentheses (like `(escaped text)`).

Grammar of Python is defined in [2] (version 2.7.6) and [3] (version 3.3.5).

1.2.2. Evaluation

Python has primitive types for *integers*, *booleans*, *floats*, *complex floats*, *strings* and *None*. Remaining types are function types and object types. Object types are defined as records containing mapping from labels into values (of type that can dynamically change). Types and type checking are described in detail in paper [1]. As the topic of type checking or type inference is not a subject of this paper, we will use Lucretia as dynamically typed language.

Variables are defined by `let in` clause and cannot be reassigned. `let in` evaluates its variable and then its body in environment with that variable's computed value.

Operators work exactly like functions (and can be implemented as built-in functions). Semantics of most operators is standard. Logical operators `&&`, `||` do not exist, because their usual semantics is short-circuit. They can be replaced by `if` expressions. Function and operator arguments are evaluated from left to right before evaluation of that function/operator. Function call evaluates expression that is body of the function and returns its value. This expression is evaluated in environment from function definition (function closure), but with

```
Expr             ::= ConstInt | ConstBool | ConstFloat | ConstImaginary |
                     ConstString | ConstNone | Var | Op | New | LetIn | Prop |
                     AssignProp | IfElse | Fun | Appl | Import | Paren
ConstInt         ::= ... | -1 | 0 | 1 | ...
ConstBool        ::= True | False
ConstFloat       ::= ... | -0.5 | ... | 0.0 | ... | 0.5 | ...
ConstImaginary   ::= ... | -0.5j | ... | 0.0j | ... | 0.5j | ...
ConstString      ::= "(escaped text)"
ConstNone        ::= None
Var              ::= Ident
Op               ::= OpType ( [Expr ,] )
LetIn            ::= let Ident = Expr in Expr
New              ::= new
Prop             ::= Ident . Ident
AssignProp       ::= Ident . Ident = Expr
IfElse           ::= if Expr then Expr else Expr
IfHasAttr        ::= ifhasattr (Expr, Ident) then Expr else Expr
Fun              ::= func ( [Ident ,] ) { Expr }
Appl             ::= FunExpr ( [Expr ,] )
FunExpr          ::= Var | Prop | Fun | Paren
Import           ::= import ( ConstString )
Paren            ::= ( Expr )
Ident            ::= (identifiers)
OpType           ::= + | - | * | / | == | != | ! | ...
```

Figure 1.1: Grammar of Lucretia

current heap state. Function definition does not contain function names and is not recursive by itself. Recursion can be achieved using closure and objects as described in the next chapter.

The expression `new` creates a new object onthe heap, with no attributes (an empty record). Property assignment adds or updates value in a given attribute. There is no way to remove an object or its property. There is also no way to iterate over all properties or make a copy of an object aside from doing it manually using `ifhasattr` on each possible label. Objects are passed by reference, i.e. only their location on heap is copied. The conditional `ifhasattr` takes an expression returning an object and a label. It is evaluated to one of its branches depending on whether object contains property with the given label. Passing a non-object value to `ifhasattr` results in error. This will be important later on when we will be dealing with objects.

When using operators or `if`, each type can be used as *boolean*. Objects and functions are `True`, `None` is `False`, *strings* are `True` iff not empty and numeric values are `True` iff equal to zero. Operators are polymorphic and can take different primitive types of the same kind. *Boolean* values can be converted to numeric values (as 0 or 1) and numeric values can be converted to more generic ones (*bool* $\rightarrow$ *int* $\rightarrow$ *float* $\rightarrow$ *complex*). Note that *None*, object and function values cannot be used as numeric values in operators. Comparison operators can be used on all types convertible to numeric and on *strings* (with lexicographical order). Equality operators (`==` and `!=`) can be used on any two objects of any kind, though comparison of functions always yields `False` and two objects are considered equal iff they have the same heap location. Equality operators also allow for conversion. Strong equality operators were introduced (`===` and `!==`) to allow for checking of equality without applying conversion.

Note that Lucretia has the following limitation: we cannot detect type of a variable, unless it is *None* or *boolean*. Equality and strong equality are only operators that evaluate without error with any argument types.

1.2.3. Expression segments

Notion of expression segments will appear in this work. Expression segments ES are defined as functions of type Expr $\mapsto$ Expr of the following structure:

$$ES = \{\lambda e.\texttt{let}\ v\ \texttt{=}\ E\ \texttt{in}\ e \colon v \in \mathrm{Ident}, E \in \mathrm{Expr}\}.$$

Expression segments form a semigroup with function composition. We can join sequences of expression segments forming a larger one. We can apply a terminating expression to expression segment. Terminating expression will be evaluated in environment with all v_i variables visible and state of heap after evaluating all E_i expressions.

Chapter 2

Interpreter

2.1. Syntax sugar

To decrease amount of written code, the following syntax sugar was introduced in Lucretia interpreter:

- `A; B` $\leadsto$ `let _ = A in B`–shortcut for sequencing operator (see section 3.2);
- `let X = A; B` $\leadsto$ `let X = A in B`–shortcut for defining a variable (see section 3.2);
- `A.B.C` $\leadsto$ `let ab = A.B in ab.C`–shortcut for multiple dot operators.

Note that `;` operator has lower priority than `let in`, so `let in` can be used together with it for temporary variables with small scope.

2.2. Environment

Lucretia is an imperative language. It has order of evaluation of its expressions strictly defined. Evaluation of Lucretia AST consists of evaluating expressions and changing the environment according to that order. Environment consists of two properties:

- variables defined in the current scope–their identifiers and values;
- heap–mapping from locations of objects into mappings from labels to values.

In a real implementation two more properties are gathered:

- final state–a copy of environment in which last `let in` body was evaluated (without previous final states). See section 2.3.2 for its usage;
- stack trace–for debugging purposes in case of an error.

2.3. Lucretia interactive interpreter

Default implementation of Lucretia file interpreter parses the whole file as a single expression and computes its value. The same can be done in interactive interpreter. However, it is desired to have a feature of passing defined variables and environment between evaluations of input. Usually Lucretia program's structure consists of a sequence of many `let in` expressions that have simple expressions evaluated (and sometime discarded if only side effect was relevant) in their binding and another `let in` nested in their body.

We can see that what we need a way to evaluate expression segments in Lucretia interactive interpreter. Therefore we accept two types of expressions in Lucretia interactive interpreter:

- Expr–normal expression;
- `let` Ident `=` Expr–an expression segment in which we define a variable that will be available from now on.

2.3.1. Constructing evaluated code

That means that we want to construct a sequence of nested `let in` expressions, such as:

Lucretia

```
let var1 = input1 in
let var2 = input2 in
let _ = input3 in
let currentValue = input4 in
currentValue
```

At any point of time we want to hold an expression segment being a composition of all evaluated expression segments. We also want to know what is the identifier of last computed variable (`currentValue` in example above). After each input we want to compute new expression segment, compose current one with it and apply to it last computed variable. When new input is an expression segment, we do just that. When input is just an expression `E`, we translate it into the following expression segment:

Lucretia

```
let freshVariable = E
```

`freshVariable` is a fresh variable that will be used only to display value of computed expression. The example code above would be constructed in a following interactive interpreter session:

```
> let var1 = input1
*value of input1*
> let var2 = input2
*value of input2*
> input3
*value of input3*
> input4
*value of input4*
```

2.3.2. Reusing computed environment

Obviously, the whole constructed code should not be evaluated from scratch after every input. In order to reuse previous computations, we want to save interpreter environment just before displaying current value. Since evaluating a variable does not have any side effects, we can safely replace it with new expression segment along with its variable and start evaluation from this point. This is sound because `let in` has defined order of execution–first binding expression then its body.

We do this by using final state saved in the environment (see section 2.2).

2.4. Importing files

Mechanism of importing files was added to Lucretia. The phrase `import("file/path.luc")` searches for a file `file/path.luc` and parses it into an AST. When evaluating Import expression, it simply starts evaluating parsed AST in current environment without currently defined variables (but the same heap). The fact of evaluating a different file is also noted on the stack trace.

In order to facilitate creation of Lucretia libraries, a special syntax was used for file paths. If the system environment variable `LUCRETIA_PATH` is defined, paths prefixed with `%/` search for files in a directory specified by `LUCRETIA_PATH`.

2.5. Lucretia module

We can put any kind of expression into a Lucretia file, but usually imported files will contain a module–a set of functions and variables to work with. Usual module file structure is as follows:

Lucretia

```
let module = new in
let _ = module.pow2 = func(x) { *(x,x) } in
let _ = module.var = 42 in
module
```

The result of interpreting such a file would be an object with `fun` and `var` properties, which could be used later. To use them, one has to bind imported module to a variable. Example usage of such a module is:

Lucretia

```
let module2 = new in
let module = import("path/to/module.luc") in
let someVar = 44 in
let _ = module2.varToPow2 = module.pow2(module.var) in
let _ = module2.someVarToPow2 = module.pow2(someVar) in
module2
```

When using this technique, values assigned to a module object (such as `module2.varToPow2` or `module2.someVarToPow2`) are effectively exported, while values that are only bound to a variable are not exported (such as `someVar`). Note that values bound to a variable can be still present in closure of functions defined in that module.

2.6. Lucretia library

There were 4 Lucretia library modules created in the process of developing Lucretia interpreter and converter.

2.6.1. BST

Module `%/bst.luc` contains implementation of balance-bounded trees in form of a map. Their implementation was based on paper [7]. Balance-bounded trees are binary trees with logarithmic complexity of most operations. They are used in implementation of Haskell's maps and sets. Tests of this module (along with usage examples) can be found in `%/test/bstTest.luc`. The following functions were implemented (as properties of the module object): `empty`, `null`, `insert`, `findMin`, `deleteMin`, `delete`, `member`, `lookup`, `size`, `mapKeysMonotonic`. Naming of functions and their signatures are based on Haskell's `Data.Map`.

2.6.2. IO

Pure Lucretia does not contain input/output functions. However, Lucretia interpreter always prints out computed value, so a mechanism to simulate printing was created. Module `%/io.luc` contains implementation of `print`, `println` and `printbool` functions, which append their arguments to *string* `buffer`. To create a program that "prints" some output, one can simply use those functions and return the buffer at the end.

2.6.3. Python builtins

For some of converter's functionality to work, Python's `__builtins__` module had to be partially implemented. It had to be implemented in Lucretia, because it is impossible to create classes such as `object` in Python without any library. The following was implemented in `%/builtins.luc` module:

- framework for Python functions' arguments (arguments are a linked list);
- shortcuts to call Python functions with 0-3 arguments;
- `object`, `tuple`, `function`, `type`, `dict` and `list` Python classes;
- mechanism of resolving class methods (using inheritance and `__class__` property);
- mechanism of getting function out of a Python callable (e.g. a functor).

This Lucretia module is used extensively by converted Python code.

2.6.4. Test tools

For ease of testing, test tools were created in the module `%/test/tools.luc`. It contains functions `testSuite` for starting a test suite, `test` being an assertion and `testResults` returning aggregated test results along with information about success. For each assertion `0` (failure) or `1` (success) is present in the value returned by `testResults`.

Chapter 3

Converter

3.1. Overview

Let us describe conversion of Python to Lucretia. We take Python's abstract syntax tree as input and output Lucretia's abstract syntax tree. This conversion is done by recursively converting Python AST nodes with the exception of sequencing operator (see section 3.2). This chapter will only contain informal description of techniques used to convert specific Python AST nodes in the form of examples of original and converted Python code. Examples of techniques will be usually isolated from other techniques. This is needed, because many of them add large amount of Lucretia code, reducing its readability. Full specification of the conversion is the converting program itself.

Python module will be converted to Lucretia module (see section 2.5). Property names of Python objects will be preserved when possible, such as module properties (i.e. top-level definitions), class methods and variables or names of local variables in current scope object (viewable by `locals()`). To prevent collision of names between converted Python names, including names with "private" naming (prefixed by `_`) or special naming (prefixed by `__`), we prefix properties of Lucretia-specific internals with `__luc_`.

To implement more complicated functionality (especially classes), additional Lucretia code is needed. It was extracted to `%/builtins.luc` file (see section 2.6.3 for short list of contents). This file is imported to each converted Python module and is supposed to partially implement Python's `__builtins__` module.

In the description of conversion below, names in the form of a a single capital letter (e.g. `X`) mean any Python expression or statement, while names containing a capital letter followed by apostrophe (e.g. `X'`) mean Lucretia expression or expression segment (described in section 3.2) to which original Python statement was converted. Names in the form `<nameWithDescription>` will also mean any expression that was converted to Lucretia.

3.2. Sequencing operator (;)

Python statements change the environment, so a way to pass it to subsequent statements must be created, i.e. a sequencing operator. Sequencing operator can be implemented through `let in` with a fresh variable, as shown in Figure 3.1.

Sequencing operator defined in this way modifies and passes environment to the second statement. Taking into account that Lucretia program is a single expression, we can convert a Python expressions to Lucretia expressions, but Python statements have to be converted to functions of type $Expr \mapsto Expr$–expression segments described in section 1.2.3.

Python	Lucretia
X ; Y	**let** _ = X' **in** Y'

Figure 3.1: Sequencing operator

Subsequent expression segments can be connected by function composition, giving expression segment for two or more statements. Eventually an argument has to be applied to expression segment. This argument depends on the context in which a sequence (block) of statements was translated. It is the variable that is returned from Lucretia expression in environment produced from executing all statements in the block. In case of translating a Python module, this module's object would be returned. For function body, it would be returned value. However, most blocks of statements are not supposed to "return" any value. In those cases, we will simply use `None`.

3.3. Variables and scope

First approach to converting Python variables could be implemented in the following way:

Lucretia
let varName = <varExpr> **in** S'

This solution has some problems though. Consider Python code shown in Figure 3.2. Converted expression `y = x` would have to be in a body of appropriate `let in` operator to have `x` bound accurately to `2` or `3`. It cannot be in both `let in` expressions, so it would have to be duplicated. The size of converted code would rise exponentially with number of `if` statements.

To resolve this, object containing local variables is created for each scope and Python local variables are converted to this object's properties. Let's call this object a scope object, as it is a direct translation of how scopes and namespaces work in Python. At any point, we have stack of scope objects, accessible through `let in` statements that define their range and give access to variables. This method also removes problem with changing values of variables. Scope object's properties can have their values (or even types) changed. Another advantage of scope objects is that location of a scope object on heap is constant and available in environment, so they work well with naturally translated function closures. Basic usage of scope objects is presented in Figure 3.2.

```
Python                   Lucretia
---------------------------------------------------------------------
x = 1                    let locals = new in
if x == 1:               let _ = locals.x = 1 in
  x = 2                  let _ =
else:                      if ==(locals.x, 1) then
  x = 3                      locals.x = 2
y = x                      else
                             locals.x = 3 in
                         locals.y = locals.x
```

Figure 3.2: Scope object and variables

3.4. Module

A Python file consists of a module in form of a block of statements. When imported or ran, it is executed and all global definitions become properties of the module object. In Lucretia, such a module object would be a scope object. The module would be defined in a `let in` expression returning its scope object. Also, Lucretia's equivalent of `__builtins__` module is imported by default.

```
Python                   Lucretia
---------------------------------------------------------------------
x = 42                   let module = new in
                         let _ = module.__builtins__ =
                           import("%/builtins.luc") in
                         let _ = module.x = 42 in
                         module
```

Figure 3.3: Module

This idea would work well also with basic importing of converted Python modules, as they can be just assigned to a variable and their functions can be used in a similar fashion as in Python with fully qualified names. However, implementing real importing would need more work, since in Python modules are cached, there are many places in which a module is searched and there are many mechanisms that are ran while importing a module.

3.5. Functions

In Lucretia, functions are created through `func` expression and are unnamed unless used together with `let in` expression. Functions may take arguments and return a value computed in their body. In Python, functions are named and when defined they become a local variable in the current scope. Functions have their own scope for local variables.

If no value is returned explicitly with the `return` statement, the `None` value is returned. Note that the `return` statement greatly modifies execution of statements–it effectively changes all subsequently evaluated expressions into nops. Converting the `return` statement is a crucial point of the conversion. One method to do it is by using Lucretia's mechanism of returning values–simply by applying converted returned expression to expression segment of the rest of the function body. However, there are two main problems with that approach: dead code after the `return` statement and conditional statements. Removing dead code is possible by optimizing out subsequent calls. Conversion of conditional statements is simple in design–to implement them one has to duplicate all code following the conditional, append each copy to statement blocks of each conditional branch and then translate the whole statement block as Lucretia `if else` expression. As long as each branch either always returns or never returns, result with optimized out dead code would be of linear size with respect to input size. Otherwise, resulting code could grow exponentially with number of `if` statements, which is unacceptable. This conversion, along with example showing exponential growth of output code, is present in Figure 3.4. In this example the `E` statement had to be duplicated, because `E` could be evaluated in two different contexts. Exponential growth would occur if `E` contained more similar nested `if` statements.

```
 Python                       Lucretia
-----------------------------------------------------------------------
def foo():                    module.foo = func() {
  if A:                         let locals = new in
    if B:                       if A' then
      return                      if B' then
    else:                           None
      C                           else
  else:                             let _ = C' in
    D                               E'
  E                             else
                                  let _ = D' in
                                  E'
                              }
```

Figure 3.4: Failed approach to implement `return` statements

Because of that, Python's mechanism of returning values from functions cannot directly translate to Lucretia's. However, it can be simulated.

3.6. Simulation of `return` and simplified exceptions

The idea of simulating `return` is very similar to simulating exceptions (and also other mechanisms like `break` or `continue` in a loop), so we will describe those two mechanisms here at once. In operational semantics, `return` and exceptions could be introduced by adding operational semantics variable f determining control flow:

$$f \in \{Normal\} \cup \{Returned(v) \colon v \in Values\} \cup \{Raised(v) \colon v \in Values\}.$$

Original operational semantics for Lucretia is in paper [1]. Following their notation, let σ denote the state of the heap and $E\langle x \rangle$ denote executing expression x in context E. To introduce control flow mechanisms, all original Lucretia operational semantics rules could be rewritten by adding variable $f = Normal$ beside heap σ and context E. Semantics of statements `return` and `raise` could be introduced in the following way:

$$\begin{aligned}
&\sigma, f = \mathit{Normal}, E\langle \text{return } v\rangle \rightsquigarrow \sigma, f = \mathit{Returned}(v), E\\
&\sigma, f = \mathit{Normal}, E\langle \text{raise } v\rangle \rightsquigarrow \sigma, f = \mathit{Raised}(v), E\\
&\sigma, f \neq \mathit{Normal}, E\langle e\rangle \rightsquigarrow \sigma, f, E\\
&\sigma, f = \mathit{Normal}, E\langle e[x_1 := v_1, ...]\rangle \rightsquigarrow \sigma, f = \mathit{Returned}(v), E' \Rightarrow\\
&\qquad \sigma, f = \mathit{Normal}, E\langle \text{func}(x_1, ...)\{e\}(v_1, ...)\rangle \rightsquigarrow \sigma, f = \mathit{Normal}, E'\langle v\rangle\\
&\sigma, f = \mathit{Normal}, E\langle e_1\rangle \rightsquigarrow \sigma, f = \mathit{Raised}(v), E' \Rightarrow\\
&\qquad \sigma, f = \mathit{Normal}, E\langle \text{try } \{e_1\} \text{ except } (x)\ \{e_2\}\rangle \rightsquigarrow \sigma, f = \mathit{Normal}, E'\langle e_2[x := v]\rangle\\
&\sigma, f = \mathit{Normal}, E\langle e_1\rangle \rightsquigarrow \sigma, f = \mathit{Normal}, E' \Rightarrow\\
&\qquad \sigma, f = \mathit{Normal}, E\langle \text{try } \{e_1\} \text{ except } (x)\ \{e_2\}\rangle \rightsquigarrow \sigma, f = \mathit{Normal}, E'\rangle
\end{aligned}$$

Introducing those semantics would make typing rules more complex and less maintainable as language evolves. Instead of doing this, we can simulate them by adding a variable that will simulate f. Then we can translate `return`, function application, `raise`, `try` and simplified `except` into setting appropriate values of f and adding `if` expressions to change subsequent expressions into nops when $f \neq \mathit{Normal}$. In case of `except` statement, it still would not be not equivalent to Python statement, because classes, `BaseException` class and matching types would have to be introduced first.

Note that in case of `return`, the state of control flow must be kept only inside the function, so it could be a local variable. In case of exceptions, it has to be a globally accessible variable (for the whole program, not only a single module), because exceptions can be thrown and never caught. In this work we will only introduce translation of `return`, `break` and `continue`. Full translation of exceptions is more complicated and will be left for further research. Examples are shown in Figures 3.5 for `return` and 3.6 for simplified `try-except`. Adding this functionality enlarges code linearly by a constant factor, but it is still better than exponential growth presented in the previous section. Note that not each instruction has to be wrapped in `if`–it is enough to do this to each simple block, optimizing size of converted code.

```
Python                  Lucretia
--------------------------------------------------------------------------
def abs(x):             module.abs = func(x) {
  if x >= 0:              -- function prologue
    return x              let locals = new in
  return -x               let _ = locals.returned = False in
                          -- default value returned by functions in
                          -- Python is None
                          let _ = locals.returnValue = None in
                          -- function body
                          let _ =
                            if x >= 0 then
                              let _ = result.returned = True
                              let _ = locals.returnValue = x in
                              None
                            else
                              None in
                          let _ =
      -- performing instruction only when
      -- function has not returned yet
      if !result.returned then
        let _ = locals.returned = True
        let _ = locals.returnValue = -(x) in
        None
      else
        None in
    -- function epilogue
    locals.returnValue
}
```

Figure 3.5: Implementing `return` through a flag

Simplified Python	Lucretia

```
try:
  x = 42
  raise 42
  y = 0
except e:
  y = e
```
	```
	let ex = new in
	let _ = ex.thrown = False in
	let module = new in
	let _ =
	  if !ex.thrown then
	    let _ = module.x = 42 in
	    let _ = ex.thrown = True in
	    let _ = ex.thrownValue = 42 in
	    None
	  else
	    None
	let _ =
	  if !ex.thrown then
	    let _ = locals.y = 0 in
	    None
	  else
	    None in
	let _ =
	  if ex.thrown then
	    let _ = ex.thrown = False in
	    let e = ex.thrownValue in
	    module.y = e
	```

Figure 3.6: Implementing simplified exceptions through a global flag

3.7. Function closures

Function closures are defined naturally, since they exist in Lucretia and all objects visible at function definition scope are in function environment. Values of non-local variables (e.g. locations of scope objects) remain constant. Figure 3.7 illustrates a function that returns a function closure which counts the number of its evaluations.

Python 3

```
def makeCounter():
  i = 0
  def counter():
    nonlocal i
    newI = i + 1
    i = newI
    return i
  return counter
```

Lucretia

```
module.makeCounter = func() {
  let locals = new in
  let _ = locals.i = 0 in
  let _ = locals.counter = func() {
    let internalLocals = new in
    let _ = internalLocals.newI =
      +(locals.i, 1) in
    let _ = locals.i = internalLocals.newI in
    locals.i
  } in
  locals.counter
}
```

Figure 3.7: Conversion of function closure

3.8. Recursive functions

As written in "Examples" section of paper [1], `let in` is not recursive, but a function can be made recursive by accessing itself from its closure. That means it has to be a property of an object. In Python all functions are potentially recursive. Because all named Python variables (including functions) are translated into properties of a scope object in Lucretia, they are also potentially recursive. Example of a recursive function is in Figure 3.8.

Python

```
def fib(n):
  if n <= 2:
    return 1
  else:
    return \
      fib(n-1) + \
      fib(n-2)
```

Lucretia

```
let module = new in
module.fib = func(n) {
  if <=(n, 2) then
    1
  else
    +(module.fib(-(n, 1)), module.fib(-(n, 2)))
}
```

Figure 3.8: Translation of a recursive function

3.9. The `while` loops

The `while` loops can be converted to Lucretia by creating and evaluating a recursive function. Because local variables have to stay accessible and assignable, original scope object should be used in that function instead of creating a new one. Crucial part of implementing `while` (and `while-else`) is proper handling of `break`, `continue` and `return`. The`break` statement should stop evaluation of subsequent expressions, stop recursion and prevent invocation of the `else` statements block. The `continue` statement should stop evaluation of subsequent expressions, but invoke the function recursively at the end. The `return` statement should stop evaluation of all subsequent expressions, stop recursion, set `returned` flag and prevent evaluation of the `else` statements block. If `break` and `return` are not executed inside the loop, the `else` statements block should be evaluated. In order to keep the state of control flow, we allow variable f (see section 3.6) to have the following values:

$$f \in \{Normal, Break, Continue\} \cup \{Returned(v) \colon v \in Values\}.$$

In order to implement this behavior, we create a recursive function that evaluates condition expression and `while` body. Just like with `return`, we create a new local variable for keeping the state of loop and check if $f = Normal$ before evaluating each statement (i.e. that both `returned` is `False` and the state of the loop is normal). We make this recursive function return a *boolean* that is `True` when `else` statements block should be executed. Note that `while` loops may be nested, so we might have to keep track of multiple unique flags at the same time. Figures 3.9 and 3.10 show this conversion for `while` with `continue` and with `else` and for `while` with `return` respectively.

Python

```
x = 1
while x > 0:
  x = x + 1
   if x < 3:
    continue
  x = x - 5
else:
  x = x + 1
```

Lucretia

```
let module = new in
let _ = module.x = 1 in
let _ = module.whileFun = func() {
  if >(module.x, 0) then
    -- creating while state variable
    -- initialized with "normal" (0)
    let whileState = new in
    let _ = whileState.state = 0 in
    let _ = module.x = +(module.x, 1) in
    let _ =
      if <(module.x, 3) then
        -- setting variable to "continue" (2)
        let _ = whileState.state = 2 in
        None
      else
        None in
    -- checking if state is normal before
    -- executing assignment
    let _ =
      if ==(whileState.state, 0) then
        let _ = module.x = -(module.x, 5) in
        None
      else
        None in
    -- checking if "break" (1) occured
    if ==(whileState.state, 1) then
      False
    else
      module.whileFun()
  else
    True
} in
```

```
let _ =
  if module.whileFun() then
    let _ = module.x = +(module.x, 1) in
    None
  else
    None in
module
```

Figure 3.9: Conversion of the `while` statement with `continue` and `else` statements

Python	Lucretia

```
def foo():
  x = 1
  while True:
    x = x * 2
    if x > 42:
      return x
    x = x + 1
  else:
    return 1
  return 0
```

```
let module = new in
let _ = module.foo = func() {
  let locals = new in
  let _ = locals.returned = False in
  let _ = locals.returnValue = None in
  let _ = locals.x = 1 in
  let _ = locals.whileFun = func() {
    if ==(locals.returned, False) then
      if True then
        -- creating while state variable
        -- initialized with "normal" (0)
        let whileState = new in
        let _ = whileState.state = 0 in
        let _ = locals.x = *(locals.x, 2) in
        let _ =
          if >(locals.x, 42) then
            -- returning from function
            let _ = locals.returned = True in
            let _ = locals.returnValue =
              locals.x in
            None
          else
            None in
        -- checking if state is normal and
        -- function has not returned before
        -- executing assignment
        let _ =
          if ==(locals.returned, False) then
            if ==(whileState.state, 0) then
              let _ = locals.x =
                +(locals.x, 1) in
              None
            else
              None
          else
            None in
```

```
              -- checking if "break" (1) or return
              -- occurred
              if ==(whileState.state, 1) then
                False
              else
                locals.whileFun()
            else
              True
          else
            None
        } in
        let whileResult = locals.whileFun() in
        -- after invoking "while", check if function
        -- has returned before proceeding with "else"
        let _ =
          if ==(locals.returned, False) then
            if whileResult then
              let _ = locals.returned = True in
              let _ = locals.returnValue = 1 in
              None
            else
              None
          else
            None in
        let _ =
          if ==(locals.returned, False) then
            let _ = locals.returned = True in
            let _ = locals.returnValue = 0 in
            None
          else
            None in
        locals.returnValue
      } in
      module
```

Figure 3.10: Conversion of the `while` statement with the `return` statement

3.10. Short-circuit operators

Short-circuited Python operators `and` and `or` are implemented through `if` expression. In Python those operators have special semantics. The `and` operator returns first argument if it converts to *boolean* `False`, second otherwise. The `or` operator returns first argument if it converts to *boolean* `True`, second otherwise. Those operators behave as usual for *boolean* values, but also allow special constructions like the following:

Python

```
tryToGetData() or defaultValue
doSomeOperation() or exit(1)
```

Conversion of those operators is shown in Figures 3.11 and 3.12.

Python	Lucretia

`A and B`	
	```
	let a = A' in
	if a then
	  B'
	else
	  a
	```

Figure 3.11: Conversion of the `and` expression

Python	Lucretia

`A or B`	
	```
	let a = A' in
	if a then
	  a
	else
	  B'
	```

Figure 3.12: Conversion of the `or` expression

3.11. Classes

3.11.1. Overview

Following Python's idea, a class is just a named scope object (class objects) and a block of code that is executed inside it. While executing this block of code, local variables are created. Those variables are later accessible by using a dot operator on a class object. However, there are differences between class objects and just any dictionary-like objects. For purpose of this work, those are:

- Class objects are callables, i.e. objects that can be invoked using `()` construction. Invoking a class object will construct an instance of this class using `__new__` and `__init__` functions. An instance of a class has its `__class__` property set to that class;
- Class instances can resolve methods and variables of their `__class__`;
- Methods resolved through `__class__` property have object instance prepended to their list of arguments;
- Classes have a mechanism of multiple inheritance and implement a method resolution on it. For the purpose of this work, only a single inheritance was implemented, although extending it is a matter of rewriting a single Lucretia function. This mechanism is explained in section 3.11.8;
- Classes can define semantics of most operators used on their instances. In this work, only `__call__` operator was implemented, as it contained one of the most complex and generic semantics of all operators. Its semantics is described in detail in section 3.11.10.

Python to Lucretia converter implements those functionalities. Most of Lucretia code connected to converted classes is contained in `%/builtins.luc` module, while directly converted code contains mostly usage of shortcut functions. In subsequent sections we will describe changes to converter that classes have brought.

3.11.2. The core classes

To start doing anything with Python classes, one has to have at least few other classes defined:

- `object`–it is a class that usually lies in the root of the inheritance tree. Amongst other methods, it contains implementation of `__new__` (producing an instance of a given subclass) and `__init__` (so that `__init__` would always resolve to something). Both of those methods are called in a constructor;
- `type`–a metaclass that is a class of most classes. It is its own instance. It manages some internals connected with classes;
- `tuple`–it is the class of `__bases__` property of every class;
- `function`–each class that does anything useful contains a function, and functions are objects. Note that functions may be objects of different classes in CPython, e.g. function, bound method or built-in function.

Those 4 classes cannot be defined in pure Python (at least without hacks), as they form a cycle. Most classes are based on `object`. Each class has `__bases__` property of type `tuple` and `type` property usually equal to `type` class. All classes that do something useful contain functions of `function` or similar class. In other words, most classes depend on those 4 classes and those 4 classes depend on each other.

Those classes can be created directly in underlying language, which is Lucretia in our case. The key is to implement them at once without using functionality that was not evaluated yet. It can be done by defining the objects without their properties first and using them safely as long as referencing their properties is not needed. This was done in the implementation of the `__builtins__` module. List of implemented `__builtins__` classes with their Python interfaces (being subsets of their original Python interfaces) is shown in Figure 3.13.

In current implementation, the whole mechanism of metaclasses was ignored and definition of class creates a class object directly, instead of invoking `type` constructor. The `object`'s `__new__` method is the one that can actually create new objects (and set their class and other metadata). The `type` class is a class without any real functionality. The `tuple`, `dict` and `list` classes are container classes based on the balanced BST from the module `%/bst.luc`. In the `dict` class implementation, BST is used directly, while in `tuple` and `list` implementations it is used with consecutive *integers* as keys and only because tables with constant access time and arbitrary number of items have not been introduced to Lucretia. As tuples are not mutable structures in Python, `__luc_append` internal Lucretia function was introduced to initialize them. The `function` class had to be created, because `__call__` method (that allows for instances of a class to be used as a functor) is supported by the current converter. There had to be a way to distinguish between normal functions and functors. The only way to do that would be by `ifhasattr`, but then `ifhasattr` works only for objects.

3.11.3. Function objects

With introduction of classes, functions were changed into function objects. Function objects are instances of the `function` class. The underlying Lucretia function is present in their

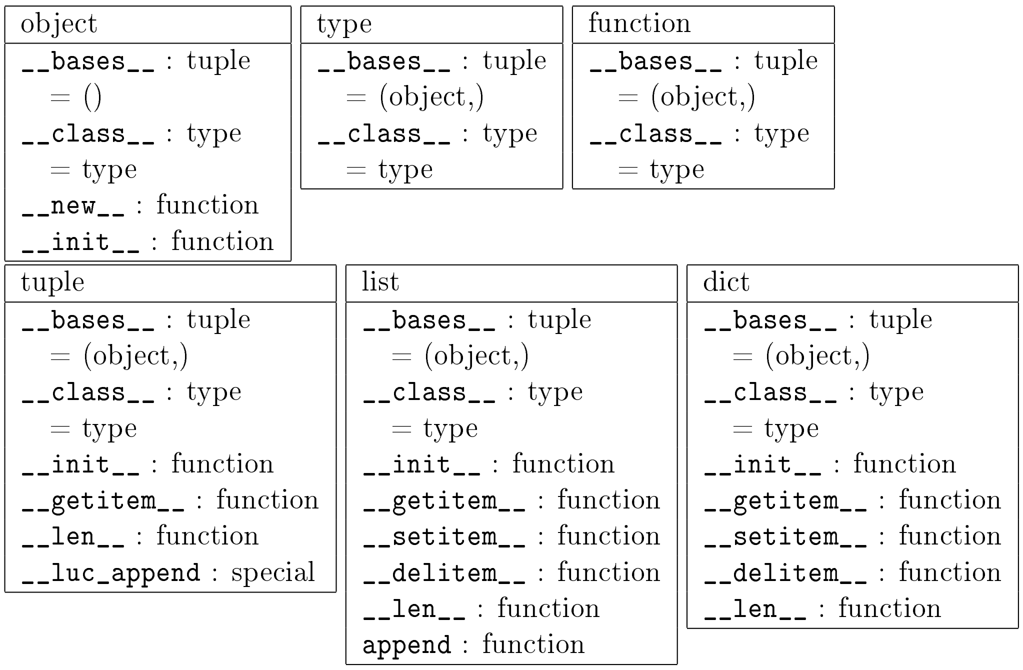

Figure 3.13: Implemented Python classes

`__luc_call` property. Those functions use the method of passing arguments described in Section 3.11.4.

The shortcut to create a function object is `__luc_mkFunction`. It returns object passed in the argument for the purpose of assigning it to a variable. It can be used in the way shown in Figure 3.14 to add an identity function `id` to scope object `scope` in a concise way and at the same time create the `id` variable with it (to be used in manually written Lucretia code).

Lucretia

```
let scope = new in
let builtins = import("%/builtins.luc") in
let id = builtins.__luc_mkFunction(scope.id = new, func(args) {
  let firstArg = builtins.__luc_takeFirstArg(args) in
  firstArg
}) in
scope
```

Figure 3.14: Usage pattern of `__luc_mkFunction`

This code both assigns a new object to the scope object property and initializes it with a function object.

3.11.4. Function arguments

Lucretia has only functions with the constant number of arguments, known also statically at the point of call. In Python, there are different constructions that allow for changing number of arguments, out of which the most important one is `self` added to class methods. When a method is called on an object and this method is a function belonging to object's class, not the object itself, the object is prepended to the list of arguments, commonly known as `self` argument. The problem is that when an object contains a function property directly (i.e. not just in its `class`), the `self` argument is not prepended. That means that without knowledge about object properties at compile time, we cannot even compute number of function arguments. Therefore, we cannot use Lucretia functions directly.

To solve this problem, converted Python functions take a single argument instead. It contains an unidirectional linked list of arguments. More specifically, this argument contains properties `first` and `last` pointing at argument nodes and used for fast prepending and appending to the list. Each node contains `value` and `next` properties. `None` works as the null pointer.

Amount of code to construct argument object is large, so there were library functions created to reduce it. The first one consists of functions in `%/builtins.luc` module that help in creation and usage of this list:

- `__luc_mkArgs`–creating new list for a specific function (and prepending `self` in some situations);
- `__luc_appendArg`–appending a single argument;
- `__luc_appendArgs`–appending a whole arguments list;
- `__luc_hasArg`–checking if a list is not empty;
- `__luc_takeFirstArg`–removing and returning first argument from the list.

For manual usage this was still a lot of code, so there were shortcut functions created for invoking converted Python functions with small number of arguments: `__luc_callWithNoArgs`, `__luc_callWith1Arg`, `__luc_callWith2Args` and `__luc_callWith3Args`. Example usage is shown in Figure 3.15.

Lucretia

```
let scope = new in
let builtins = import("%/builtins.luc") in
let _ = builtins.__luc_mkFunction(scope.mul = new, func(args) {
  if builtins.__luc_hasArg(args) then
    let arg = builtins.__luc_takeFirstArg(args) in
    let scopemul = scope.mul in
    *(arg, scopemul.__luc_call(args))
  else
    1
}) in
let num42 = builtins.__luc_callWith3Args(scope.mul, 2, 3, 7) in
let args = builtins.__luc_mkArgs(scope.mul) in
let _ = builtins.__luc_appendArg(args, 2) in
let _ = builtins.__luc_appendArg(args, 22) in
let scopemul = scope.mul in
let num44 = scopemul.__luc_call(args) in
+(num42, num44) -- == 86
```

Figure 3.15: Usage of the function arguments object

Note that this way we created a function working with variable number of arguments. Also note that to invoke `scope.mul.__luc_call` in pure Lucretia we had to extract `scope.mul` first, as dot operator works only on variables (see Lucretia grammar in section 1.2.1). This became a problem as more objects were used in Lucretia, so syntax sugar for multiple dot sequences was created (described in section 2.1).

3.11.5. Object tags

Three kinds of objects are callable: normal functions, functors (class instances with `__call__` class method defined) and class objects (as constructor). Also, semantics of the dot operator on a class (with its `__bases__`) and instance (with only `__class__` defined) are different. To help Lucretia's `builtins` library to distinguish between those cases, the following tags were added to some objects:

- `__luc_functionTag`–a tag for normal function objects created by executing their Python definition;
- `__luc_boundMethodTag`–a tag for bound methods, i.e. methods returned by the dot operator with some of their arguments already set (e.g. `self`);
- `__luc_classTag`–a tag for class objects.

Tags are simply properties of an object with the *None* value that are checked for existence with `ifhasattr` when needed. This could have been done by simply creating appropriate classes and looking up object's `__class__` property, but this way gives some basic information about object without even executing `builtins` module and might be better for a future type checker (as it can simply look up a tag instead of analyzing the heap and classes defined there, such as `type`).

3.11.6. Implementation of normal classes and shortcuts

Classes are created in the following way:

1. A Lucretia object is created (`new`) and has its properties set: `__class__` to `type`, `__luc_classTag` to `None`, `__bases__` to an empty `tuple` object and `__luc_call` to constructor definition (calling `__new__` and `__init__`);
2. Each base class expression is evaluated (from left to right and with possible side effects) and appended to the list of base classes (`__bases__`);
3. With bare class object as the current scope object, the class statements block is executed.

The first step is initialization of a new Lucretia object. It is done by the shortcut function `__luc_mkClass` in a concise way, similarly to `__luc_mkFunction`. This is an important step and needs `tuple` class to be functional, since `__bases__` tuple is created (using `object.__new__` and `tuple.__init__`) and filled in second step. Also, since not all classes have `__init__` and `__new__` defined, the constructor uses method resolution described in Section 3.11.8. Method resolution requires for all 4 core classes (`object`, `tuple`, `type` and `function`) to be fully functional. This is one of reasons why 4 core classes have to have their own initializer.

In the second step base class expressions are evaluated. They use `tuple`'s internal method `__luc_append` to fill `__bases__` tuple. This is because `tuple` does not have a public interface to modify it. Note that base class expression can be anything, not only a class name or fully qualified name. If nothing is passed, this defaults to the `object` class from the `builtins` module.

Third step consists of evaluating the body of a class using the same mechanisms as everywhere, only that scope object is a class object now. The class object is a scope object that is a named property of another scope object that class is within, e.g. module object. When body of a class is converted, variables and functions are handled as usual–by assigning a property with their name to scope (class) object. For everything to work, each function has to use the arguments objects and be wrapped in `__luc_mkFunction`.

3.11.7. Implementation of classes `object`, `type`, `tuple` and `function`

Implementation of those 4 classes is special. They are defined at the same time. First, their class objects are created. Only then can `__luc_mkFunction` shortcut be defined (since class `function` exists, so we can set function object's `__class__` now). Then, methods of those classes are created using `__luc_mkFunction`. Next step is to initialize each class–including `__bases__` tuple, so it has to be working. The `tuple` class must initialize its own `__bases__` property. The function `__luc_mkInternalClass` is a function that can initialize those 4 classes. Its difference from the `__luc_mkClass` function is that it does not resolve `__new__` and `__init__`, but instead assumes that created class contains `__init__` directly and `__new__` only through its `object` base class. This is the case for all internally defined classes. Also, both of those functions create the constructor before initializing `__bases__` property. Creating `__bases__` means using `tuple`'s constructor, which invokes the `__init__` method from the `tuple` class and the `__new__` function from the `object` class. Both of invoked methods are of type `function`. That's why `object`, `tuple` and `function` had to be defined first. The `__class__` property of initialized functions is set to `type`, so it had to be at least created before. Since `__bases__` are created after constructor, it is safe to use `__luc_mkInternalClass` on the `tuple` class. After initializing the `tuple` class, other classes can be initialized the same way.

3.11.8. Method and variable resolution

Resolution is a Python mechanism that is part of the inheritance mechanism. In languages with multiple inheritance such a mechanism is not trivial. There has to be a way to choose correct method implementation when two or more base classes (or their base classes) define a method with signature we are searching for. In Python, since version 2.3, a C3 MRO (Method Resolution Order) algorithm is used (see [4]).

Besides finding which class a method is from, MRO algorithm has one more basic task: to find this inheritance tree first. Let us describe default behavior in Python. When this algorithm gets an object, it first tries to find searched property in the object him directly (i.e. in its underlying dictionary). If the property is not found and the object is not a class, it searches (i.e. invokes search function recursively) for it in object's `__class__`, modifying value returned by dot operator in some cases (see section 3.11.9). If object is a class and property is not found, the property is searched for in the inheritance tree of classes, i.e. by recursively searching in `__bases__` properties of classes using C3 MRO algorithm. That means that in the recursive call the property is first searched for in the object itself, then its `__class__` or base class then in class' class (usually `type`) until search hits bottom (usually `type`).

The MRO algorithm is used when dot operator is invoked. The approach to MRO in this work was simplified compared to what Python uses:

- In Python, even semantics of dot operator can be changed (see [5]) through special methods such as `__getattr__`. In this work, only default behavior was implemented;

- Python MRO algorithm produces linearization of the base classes tree in `mro` class method or `__mro__` attribute. In this work, a special function is called each time dot operator is used;
- Python implements full C3 MRO algorithm, while this work only implements basic MRO algorithm that only resolves attributes from first base class. This can be easily changed by modifying `__luc_resolveProp` Lucretia function form `builtins` module;
- There are usually maximum 3 levels of objects: an instance, its class and its class' class (which is usually equal to `type`). Though practically not used, Python allows for more levels. In this work, we assume a maximum of 3 levels and that `type` class is always at deepest level. In other words, we stop search for the property at object's class level.

Implementation of the dot operator is contained in the `builtins` module in the Lucretia function `__luc_dot`. It takes left-hand side the dot operator and property as argument and returns the resolved object or `None` if it was not found. The problem with implementing this function in a generic way is that Lucretia does not allow passing property labels as function arguments or any other kind of parameter. One way to solve this problem would be to create MRO function for each property label used in the program. This would needlessly produce very large amount of code. Approach taken in this work was to create two special functions, property checker and property getter and passing them to functions that need a property label.

A property checker is a function that takes an object as argument and returns `True` when that property exists inside the object, `False` otherwise. A property getter returns that property. For property named `X`, those functions are defined and used as follows:

Lucretia

```
func(obj) {
  let XChecker = func(obj) { ifhasattr (obj, X) then True else False } in
  let XGetter = func(obj) { obj.X } in
  if XChecker(obj) then
    doSomethingWithProperty(XGetter(obj))
  else
    doSomethingElse()
}
```

Note that a single function could be implemented for both checking and getting a property, but it would have to return some special variable if it was not found. Also, it would have to be distinguishable by Lucretia from other types and not possible to create through conversion. To remove unnecessary complexity, two functions are used.

Besides resolving an object, the dot operator has one more function when class method was resolved–returning method bound to an object instead of the function itself. It is explained in Section 3.11.9

3.11.9. Bound methods

When Python's dot operator resolves a function and it does not find that function in an object directly, but in its `__class__` instead, it binds the left-hand side object of the dot operator as the first argument of that function. It is commonly known as `self` and a class function with bound argument is called a bound method. It is worth noting that binding happens in the dot operator, not in the call itself. That way we can assign a result of a bound method to a variable and call it somewhere else with first argument still being bound.

The whole mechanism in which the arguments object is passed to a function (described in Section 3.11.4) was created for this purpose–to allow for prepending `self` to the list of arguments, while not knowing the number of function arguments at compile time at the point of call. Bound method is a Lucretia object that has three attributes:

- `__luc_self`–the bound object;
- `__luc_call`–the underlaying Lucretia function, taken from the original function's object `__luc_call` property;
- `__luc_boundMethodTag`–a tag to distinguish bound method from other callable objects.

3.11.10. Callables

There are four types of objects that can be called, i.e. call operator `()` can be used on them: functions, bound methods, classes (as constructors) and functors (instances of classes with `__call__` method defined). First three kinds of those objects have their own tag (see section 3.11.5), therefore can be distinguished by `ifhasattr`. Functors can be distinguished by having `__call__` property and not having any of previously mentioned tags.

Calling functions and class constructors is simple–the code lies in their `__luc_call` property. For bound methods, additionally `self` argument has to be prepended. This is done automatically in `__luc_mkArgs` function that takes one of those callables as argument.

Calling functors is a bit more complicated. Their `__call__` function is called instead and first argument is set to the functor object itself, followed by arguments from call invocation. This behavior works intuitively as long as we use functors as functions. When we use functors as class methods, `self` argument is not prepended–it is set to the functor object instead. In other words, resolved value is treated as an object, not as a method that would be automatically bound by the dot operator. The same behavior can be observed when creating a functor with the `__call__` property set to another functor–the first argument is set to the deepest functor object and the rest of arguments is taken from the call invocation. This can be confusing, as semantics of bound methods and functors is not about prepending an object to the list of arguments, but rather about prepending up to one argument. For clarity, Python's documentation (see [5]) describes usage of functor `x` in call `x(arg1, arg2, ...)` as a shorthand for `x.__call__(arg1, arg2, ...)`, which interpreted textually for nested functors means that in fact only the last functor object should be prepended to the list of arguments (using only bound method semantics).

Chapter 4

Program development and usage

4.1. Overview

This work resulted in creating a program `lucretia` with the command-line interface, which has the following functionalities:

- interpreter of Lucretia files (`.luc`);
- Lucretia interactive interpreter with additional debug commands;
- conversion of Python file into Lucretia;
- interpreter of Python files through Lucretia converter;
- self-test.

It also supports importing Lucretia files from Lucretia library directory. Basic Lucretia library was implemented.

4.2. Technology

The `lucretia` program was written in Haskell (using ghc 7.6.3 compiler). The project is in the form of cabal package and depends on the following packages (from the Hackage package archive):

- `language-python`–the Python 2 and Python 3 parser;
- `parsec`–the generic parser package used to parse Lucretia;
- `HUnit`–the testing framework based on unit tests;
- `QuickCheck`–the testing framework based on generating multiple data sets and checking if given properties hold;
- `options`–the package providing parsing of command-line options;
- `placeholders`–the package used to mark unimplemented fragments of the program, while allowing for execution of its implemented part;
- `base`–the base ghc package;

- `containers`–the package providing various data structures;
- `directory`–the package used for listing directory contents;
- `filepath`–the package used for constructing file paths in a cross-platform way;
- `mtl`–the package providing various monads.

Project was written and tested mainly on Windows 7 with The Haskell Platform. IDE used to write it was Leksah 0.12.0.3. The Leksah IDE is also available for Linux. The project was also tested on a Debian 7 with the ghc compiler and the cabal program from Debian's repository.

4.3. Architecture

The `lucretia` cabal package is split into modules. The modules could be categorized thematically into Lucretia interpreter, Python converter, testing and utility.

4.3.1. Lucretia interpreter modules

- `AST`–defines Lucretia AST and structures like parsed Lucretia module with names. For optimization purposes variable and property label identifiers are saved in AST as numbers and mappings from variable and property label numbers into their names are held separately;
- `ASTFunctions`–contains functions connected to AST, such as traversing AST or updating Lucretia module group structure;
- `Interpreter`–contains implementation of Lucretia AST interpreter;
- `LucretiaParser`–parsing Lucretia into AST;
- `ParserCommon`–common parser functions, mainly fixing annotations about position in the source code;
- `RunEnvironment`–contains definition of Lucretia runtime state (with such data as heap and mapping from variables to values);
- `InteractiveInterpreter`–implementation of Lucretia interactive interpreter;
- `RuntimeError`–definitions of possible runtime errors;
- `FileInterpreter`–uses `LucretiaParser` and `Interpreter` to interpret and pretty print the result of evaluation of a Lucretia file.

4.3.2. Python converter modules

- `ConvertEnvironment`–contains definition of the Python converter state (with such data as the stack of scopes and the mapping of variable names) and functions operating on it;
- `PythonFileInterpreter`–uses `PythonTestInterpreter` on a file and prints the `test` top-level variable;

- `PythonTestInterpreter`–contains function that convert a Python file, extract the `test` top-level variable and interprets the result with the Lucretia interpreter;
- `PythonFileConverter`–uses `PythonConverter` to parse a Python file, convert it to Lucretia and pretty print the AST;
- `PythonConverter`–parses and converts Python files and code (as *string*) to Lucretia AST.

4.3.3. Testing modules

- `ASTArbitrary`–defines classes to produce an arbitrary AST for testing purposes;
- `TestTools`–contains modification of `quickCheckAll` from `QuickCheck`;
- `Test`–`QuickCheck` and `HUnit` Lucretia tests, iterating over test files and testing result and some test tools.

4.3.4. Utility modules

- `Common`–contains a few constants and very short generic functions;
- `CmdLineOptions`–parses command line options;
- `Main`–contains the program main function that uses command line options to delegate work to an appropriate module;
- `PrettyAST`–pretty printing AST to readable Lucretia code with indentation;
- `PrettyError`–pretty printing errors;
- `PrettyValue`–pretty printing values, also recursive printing of object values;
- `ImportFileIO`–reading files with specified path, possibly prefixed by library prefix `%/`;
- `StringEscape`–functions to escape *strings*;
- `PrettyCommon`–common pretty printing functions.

4.4. Automated tests

In order to reduce the number of bugs, many tests were written for `lucretia`, focusing mainly on Lucretia interpreter and Python converter. There were three types of tests used in this work: `HUnit` unit tests, `QuickCheck` property assertion tests and file-based tests.

`HUnit` is unit testing framework that takes in a list of tests, evaluates them and checks assertions. `HUnit` tests can be grouped into categories, which was done in this work. Written unit tests took Lucretia AST, Lucretia code or Python code as input, evaluated, parsed and converted it as needed and then checked if evaluation was successful (or in some cases–if it failed as expected) and the evaluated value was compared with the expected one. The evaluated value was either value of the interpreted Lucretia AST, code or a value of the top-level `test` variable of a Python module. Tests were written so that only simple values were computed, i.e. not functional or object values, which would be hard or impossible to compare. Those tests checked if Lucretia and converted Python language constructs were evaluated correctly. This checked either Lucretia interpreter, interpreter with parser or Python converter with Lucretia interpreter. At first, interpreter was tested directly with Lucretia AST, but tests of Lucretia code were proven to be much more efficient to write and maintain. There were also a few tests written for the Lucretia parser itself, using function `haveSameStructure` described at the end of this section.

`QuickCheck` is a testing framework that asserts properties of a given model using random data. `QuickCheck` tests are definitions of properties that should hold (i.e. evaluate to `True`) for variables of given type. Generator for such variables has to be supplied each time. An example of such a test is testing commutativity of multiplication operator by checking if for all *integers* `x` and `y` the followingholds: `x * y == y * x`. Such a test would generate a large number of pairs of *integers* and test this property. Functions that operate on metadata of parsed files and checking Lucretia AST pretty printer were tested this way. Lucretia AST pretty printer was supposed to produce parsable Lucretia code that would parse to equivalent Lucretia AST.

Testing of language constructs was done mainly in the Haskell module. However, unit tests with code snippets contained in Haskell module were inappropriate for larger tests, especially for tests of the Lucretia library. There were file-based tests implemented that traversed the `test` directory of the Lucretia library directory and tested all files with names ending on `Test.luc` or `Test.py`. Testing a file was done by parsing it, in case of Python converting to Lucretia, evaluating and checking if result was correct (i.e. was a *string* in the correct format). For Lucretia code, evaluated value of Lucretia AST was taken directly, while for Python files `test` top-level variable was checked.

There were also various helper functions written to remove redundant code from written tests. Many of them were taking Lucretia AST or Lucretia code or Python code, parsed, converted and evaluated it as needed and checked if evaluation succeeded with the expected value (or sometimes if evaluation failed as expected). Particularly interesting helper function was `haveSameStructure`–it took two Lucretia ASTs and checked them for equivalence modulo numbering of identifiers, code annotations and small differences between *floats*. It was used for checking correctness of the Lucretia parser and the Lucretia AST pretty printer. Another interesting set of functions was `ASTArbitrary` module, which recursively generated random Lucretia AST (often incorrect) that was often large enough, but finite with probability 1.

4.5. Benchmarks

There were a few benchmarks made in order to check what optimizations are required in the future. All benchmarks were run on a Windows 7 laptop with MinGW bash, `lucretia` compiled with ghc 7.6.3, on a machine with Intel i7 2GHz processor (tests were using a single core) and 8GB RAM.

4.5.1. Benchmarks of tests

In order to use tests in such a programming development process as Test-Driven Development, they have to be quick enough to run them between implementations of very small functionalities. As described in Section 4.4, there were four types of tests run: `QuickCheck` invariant tests, `HUnit` unit tests, file-based Lucretia library tests and file-based Python conversion tests. Speed of those tests was measured by using `time` program and an average time length from 10 runs was taken. The results are visible in Figure 4.1.

Because the tests were run separately, there is an overhead of creating a process and running a short `bash` script contained in each of those results. The final result yielded that `HUnit`

Test	Time
All tests	1.93s, varies greatly
QuickCheck	0.92s, varies greatly
HUnit	0.364s
Lucretia library tests	0.69s
Python conversion tests	0.31s

Figure 4.1: Benchmarks of `lucretia` tests

unit tests were indeed fast enough, even though they contained the largest amount of tests. `QuickCheck` tests were slow, although they tested more complicated and generic functionalities. The problem with `QuickCheck` tests was that they were producing nondeterministic output. This can be perceived as a feature–different sets of data are tested each time–but because of this they were sometimes taking more time than average. This is because randomly generated test ASTs were sometimes too big. Both file-based tests were fairly slow, even though they contained small amount of tests. Most probably large amount of file-based tests would not be useful for quick testing. Appropriate method of using those tests would be selecting a small part to run while developing the program (i.e. the one developer is working on).

4.5.2. Benchmarks of the Lucretia converter

Benchmarks of the Lucretia converter were made for two purposes:

- To assess whether the current implementation of interpreter can handle computationally intensive tasks;
- To check that the output size of converted code with some problematic constructs was linear with respect to the Python code size.

There were four test suites run and statistics about time length of conversion and the number of lines of code were taken (KLOC means thousand lines of code). Each test was run 10 times and average time length was taken.

The first test suite (Figure 4.2) contained N functions computing Fibonacci numbers iteratively with fully unrolled loop. Python code length increased with rate $O(N^2)$. We can observe that size of converted code is roughly 6 time as large as input Python code and this proportion stays linear with increasing N.

The second test suite (Figure 4.3) contained $N + 1$ classes `Ai` for `i` $= 0..N$, where class `A(i-1)` was a base class of `Ai` class. Class `A0` additionally contained a method. An instance of each class was created and this method was invoked (with full usage of method resolution algorithm). Python code length increased with rate $O(N)$. Again, we could observe linearity of conversion and roughly 9 times increase in the number of converted code lines.

The third test suite (Figure 4.4) contained N nested `while` loops with `if else` and `return` statements in-between. Python code length increased with rate $O(N^2)$, but the number of lines increased with rate $O(N)$, since indents were the factor that caused $O(N^2)$ increase. We could once again observe linear increase in the number of converted code lines.

The fourth test suite (Figure 4.5) was special–it contained constant number of Python lines in which an object `x` had property `x.p`, which referenced back to `x`, allowing for chaining the dot operator N times in the following way: `x.p.p.p.p...p`. This test was chosen, because conversion of the dot operator is one of the most expensive translations in terms of amount of produced code. Since translated code had many nested `let in` expressions, it was automatically indented, producing code with $O(N^2)$ size. However, this is only a result of pretty printing. To properly measure increase in amount of produced code, the number of bytes of Python code was compared with the number of non-space bytes of Lucretia code. This proportion was growing, but was slowly starting to stabilize after around 80.0 ratio. Note that this number is increasing, because a fair amount of Python code bytes was used on the construction of the object. However, this ratio should keep growing with rate $O(\log N)$, because the names of unique temporary variables in `let in` were getting longer. This overhead could be removed by reusing temporary variable names or by simply counting AST nodes instead.

N	Conversion time t	$\frac{t}{t_{N=100}}\left(\frac{100}{N}\right)^2$	Python KLOC	Lucretia KLOC	$\frac{\text{Lucretia KLOC}}{\text{Python KLOC}}$
50	0.391s	1.31	3.9	23.4	6.00
100	1.19s	1.00	15.3	91.7	5.99
150	2.53s	0.94	34.1	205.0	6.01
200	4.45s	0.93	60.5	363.4	6.02
300	9.83s	0.92	135.8	815.1	6.00
400	17.3s	0.91	241.0	1446.8	6.00

Figure 4.2: Benchmarks of the Lucretia converter–Fibonacci numbers

N	Conversion time t	$\frac{t}{t_{N=1000}}\left(\frac{1000}{N}\right)$	Python KLOC	Lucretia KLOC	$\frac{\text{Lucretia KLOC}}{\text{Python KLOC}}$
100	0.370s	1.29	0.4	3.6	9.00
500	1.43s	1.00	2.0	18.0	9.00
1000	2.87s	1.00	4.0	36.0	9.00
2000	5.71s	0.99	8.0	72.0	9.00
5000	14.39s	1.00	20.0	180.0	9.00

Figure 4.3: Benchmarks of the Lucretia converter–large inheritance tree

N	Conversion time t	$\frac{t}{t_{N=20}}\left(\frac{20}{N}\right)$	Python LOC	Lucretia LOC	$\frac{\text{Lucretia LOC}}{\text{Python LOC}}$
10	0.178s	0.50	82	889	10.8
20	0.706s	1.00	162	1759	10.8
30	1.88s	1.33	242	2629	10.9
40	4.20s	2.97	322	3499	10.9
50	7.97s	4.52	402	4369	10.9
100	-	-	802	8719	10.9

Figure 4.4: Benchmarks of the Lucretia converter–nested `while` loops

N	Python bytes	Lucretia non-space bytes	$\frac{\text{Lucretia non-space bytes}}{\text{Python bytes}}$
10	70	2333	33.3
100	250	16733	66.9
500	1050	85321	81.3
1000	2050	171321	83.6
2000	4050	352409	87.0

Figure 4.5: Benchmarks of the Lucretia converter–sequence of properties

Tests have shown that increase in output code size is linear or at most linear-logarithmic in most sane conditions and that quadratic increase happens only because of pretty printer indents.

4.5.3. Benchmarks of the Lucretia interpreter

There were also tests made to compare the speed of the Python interpreter with the Lucretia interpreter using converted Python code. The results were that current implementation of Lucretia interpreter is not able to handle any larger tasks. It is slow, does not have a garbage collector and runs out of stack and heap memory for large computations. For some tests the decrease in speed is hard to compare, because Lucretia interpreter runs out of memory before Python code's running time stops being dominated by overhead of creating a new process and a Python interpreter instance.

There were four tests made. The first one computed 26th Fibonacci number recursively. The second one computed 50,000th Fibonacci number iteratively. The third one implemented a `Counter` class, created its object and incremented the counter 50,000 times. The fourth one contained large tree of 1000 inherited classes (the same tree as in Section 4.5.2), created an object of the class that was the furthest from the root of inheritance tree and resolved a method from the class at the root 100 times. All tests were run 10 times and average time length was taken. Results are shown in Figure 4.6.

Task	Lucretia	Python
Fibonacci recursively	24.39s	0.21s
Fibonacci iteratively	2.43s	0.16s
`Counter` class	5.81s	0.13s
Deep inheritance with method resolution	12.37s	1.32s

Figure 4.6: Benchmarks of the Lucretia interpreter

4.6. Build instructions

To build `lucretia`, one has to have installed `ghc` and `cabal` programs and run the following commands in the Lucretia project directory:

```
cabal install --only-dependencies
cabal configure --enable-tests
cabal build
```

This will install dependencies, configure the project to build together with tests and build it. To install `lucretia` as command-line utility, one has to additionally run:

```
cabal install
```

For those commands to work, `ghc` and `cabal` have to be in the `PATH` environment variable. Additionally, to build the `language-python` package, which is a dependency of `lucretia`, `happy` and `alex` parser programs have to be installed and accessible through the `PATH` environment variable.

4.7. Usage instructions

Name of program executable implemented in this project is `lucretia`. It is a command-line program. To run it correctly, the `LUCRETIA_PATH` environment variable has to be set first to the path of Lucretia library directory. If the variable is not set, importing library files and evaluating converted Python code will not work. For a list of possible command-line options, `lucretia --help` can be run. Possible options are:

- `lucretia --mode interactive` or `lucretia`–run Lucretia interactive interpreter;
- `lucretia --mode interpreted fileName.luc` or `lucretia fileName.luc`–evaluate Lucretia file and pretty print its value;
- `lucretia --mode python fileName.py` or `lucretia fileName.py`–parse, convert to Lucretia and pretty print the `test` top-level variable of a module in the given Python file;
- `lucretia --mode convert fileName.py` or `lucretia -m c fileName.py`–parse a Python file, convert it to Lucretia and pretty print Lucretia code;
- `lucretia --mode test`–run `lucretia` tests.

Chapter 5

Conclusion

5.1. Accomplished goals

The Lucretia interpreter along with a few debugging tools was finished and it is usable enough to write larger Lucretia programs with it. Thanks to displaying stacktrace and exact point of failure on errors, it is usually easy to fix bugs. However, Lucretia Parser should be improved, as its current parse error messages are confusing when for example operator is used in infix notation.

The Python to Lucretia converter is still incomplete. Framework for handling scopes, basic constructions such as functions and `while` loop were completed. The largest amount of work was spent on classes. The Python classes are still only partially supported, however most important mechanisms have been implemented (inheritance, method resolution, part of `builtins` library). Classes are central functionality in Python. They are required to implement many other features, including:

- iterables (along with `for` loop);
- exceptions (all exceptions should derive from `Exception` class in Python 3, see [6] for details);
- The `with` statement;
- importing and using fully qualified names, unless we give special semantics to dot operator;
- `*args` and `**kwargs` constructions, where `args` is an iterable and `kwargs` is of the `dict` class;
- primitives (e.g. *integers*, *strings*) being objects and having their own methods.

5.2. Production Python code support

Tests of the converter on the production code were not performed, because some very popular language constructs are not implemented, mainly exceptions (which can be thrown in many places in Python), import mechanism (most Python libraries and programs import something) and constructs for iterables, such as `for` loop or the subscript (`[]`) operator.

5.3. Future work

The `lucretia` program and the Lucretia language will be still developed in future. The goal is to be able to convert most production Python code into Lucretia and then try to statically type it. In order to do this, more Python language features have to be converted. This includes features for which classes were needed listed in section 5.1, but also:

- the importing system;
- the conversion of part of the Python standard library (partially manual);
- metaclasses;
- the rest of Python language constructs (e.g. `for`, `yield` and many operators).

Note that this list is still not full and complete Python implementation, including the Python standard library, will probably never be accomplished. Some language features are virtually never used (such as substituting the `type` class or class inherited from it as class' class). There are also language constructs which are impossible to type in general (such as `exec` and `eval`), so, aside from some special cases, converting them will not be of any benefit for type checker.

Another feature that will be desirable with introduction of the type checker is parsing Python-compatible type annotations that do not affect code execution. That way part of Python code could be typed when type inference algorithm is still not complete or cannot solve a specific case.

The intermediate goal is to implement enough Python support, so that a few chosen libraries will successfully convert into Lucretia and then start working on a basic type checker and a type inference algorithm for Lucretia to type those libraries. In the paper [1] type-checking rules for Lucretia are proposed. However they will have to be extended to support more complicated Python types (such as linked list of variable length). Type inference algorithm for Lucretia has not been designed yet.

There is also a decision to make about supported Python versions. Obviously Python 3 should be supported, as the current version of the Python language. Python 2 is still widely used, but the number of libraries ported to Python 3 is increasing. Supporting Python 2 would mean implementing old-style Python classes and other constructs that were removed from Python 3 and are deprecated in new Python 2 versions. However, those constructs are still present in old Python 2 libraries.

Bibliography

[1] Marcin Benke, Viviana Bono, Aleksy Schubert, Lucretia–ad-hoc polymorphism for scripting languages (extended version).

[2] Full Grammar specification–Python v2.7.6 documentation, `http://docs.python.org/2.7/reference/grammar.html?highlight=grammar`.

[3] Full Grammar specification–Python 3.3.5 documentation, `http://docs.python.org/3.3/reference/grammar.html?highlight=grammar`.

[4] The Python 2.3 Method Resolution Order, `https://www.python.org/download/releases/2.3/mro`.

[5] Data Model–Python 3.3.5 documentation, `https://docs.python.org/3.3/reference/datamodel.html`.

[6] Errors and Exceptions–Python 3.3.5 documentation, `https://docs.python.org/3.3/tutorial/errors.html`.

[7] Stephen Adams, Implementing Sets Efficiently in a Functional Language, CSTR 92-10, University of Southampton.