Extending a ChocoPy compiler to include lists and classes on the heap

Grane, Ellen
D17, Lund University, Sweden
el6626gr-s@student.lu.se

Alkhodary, Samer
D17, Lund University, Sweden
sa2808al-s@student.lu.se

Abstract

ChocoPy is a programming language intended to be used for learning about compilers. An existing ChocoPy compiler was extended to support heap-dynamic lists and classes stored on the heap. This was done in order to examine the suitability of the ChocoPy language to be used for learning about compilers in an advanced course. The speed of the extended compiler was compared to the standard Python interpreter and the results showed the ChocoPy compiler to be much faster. It was concluded that ChocoPy would work well for teaching more about compilers due to its many features, however providing stable bases for the students is probably needed in order to implement them, as underlying bugs can lead to many issues.

1 Introduction

ChocoPy is a subset of the programming language Python, specifically designed to be suitable to use for learning about compilers and programming languages. For example, by letting students implement a compiler for ChocoPy in a compilers course. A working ChocoPy program should be possible to execute using an ordinary Python compiler. [5]

The aim of this report is to expand an existing ChocoPy parser, that are built with JastAdd, to include the functionality to generate assembly code for classes, inheritance and heap-dynamic lists. In order to examine more advanced functionality that may be used in an advanced compilers course, some extra functionality that is not specifically part of the ChocoPy language but is included in Python will be implemented, such as heap dynamic lists and the delete function that marks objects as 'dead' in the heap and paves way for garbage collection. The compiler that was extended included an implementation for scanning and parsing of most parts of ChocoPy, and primarily missed scanning and parsing for lists and strings. It further had not yet implemented code generation for lists and strings, did not have fully functional classes and subclasses nor for-loops. [7]

The purpose of the report is further to investigate the advantages and the possible disadvantages of using ChocoPy in compiler courses and to compare it with SimpliC language that is used in EDAN65 course at LTH. ChocoPy can be expanded to cover lists, classes, inheritance and garbage collection. These features are missing from the C language which SimpliC confirms to. We also want to examine whether it is feasible to expand a ChocoPy compiler in JastAdd as a way of learning more about heaps in a compilers course. Furthermore we plan on assessing the compiling time of a few ChocoPy files using the standard Python compiler and compare it to the compiler we have extended.

Course paper, EDAN70, Lund University, Sweden
January 16, 2022.

1.1 Expansion of the ChocoPy compiler

To investigate these aspects, we decided to implement the following extentions to the already existing compiler:

- Parsing and code generation for heap-dynamic lists whose size can be changed at run time
- Fix classes so that aliasing works and implement inheritance functionality
- Add a "del" keyword that marks list and class instances as dead in the heap

2 Technical Background

In this section we will present some general information about the heap and how it can be used in assembly. We will furthermore write about classes and how they may be allocated on the heap according to relevant literature.

2.1 JastAdd

JastAdd is a system created with the intention of being used for building compilers and analysis tools. It is an extension to Java which provides features that lets the writer rewrite abstract syntax trees. It combines object-orientation, static aspects, and declarative attributes to form a powerful tool with a high support for modularity and extensibility. [2]

2.2 Heap allocation

When a computer program starts, there are usually three types of memory segments that are allocated by the operating system:

- The code segment that has the program code which is normally read-only or execute-only. The code segment is addressed by the program counter.
- The stack segment containing the stack, which is addressed by one or more stack pointers that can be altered by machine code.

- The data segment, also called the heap, which is a number of addresses to memory locations that are available for data storage.

The heap can be used to allocate dynamically sized data, meaning data whose size is decided or can be changed at run time. The operating system keeps a pointer to the address of the first available location in the heap, and the programmer can request it using machine instructions. Data can then be allocated on the heap by using the address to the first free space and subsequently increasing the pointer in order to get the next free location. [3]

In an implementation guide for generating RISC-V assembly code for ChocoPy, it is suggested use alloc() and alloc2() for heap allocation. [6] In x86 which is the assembly code used by the compiler we are extending, there exists an equivalent method called malloc() that can be imported from C. We did not use this method, as we decided to use the low level function brk() instead. The implementation guide further recommends storing classes on the heap. [6]

In Linux machines, every program gets assigned a program break which marks the end of the program's data segment. In order for the program to use more memory, the program should be moved to increase the heap size and to allows the program to utilize more memory. This can be achieved using the brk() system call which takes in the address of the new program break as an argument and returns the address of the last slot in the newly allocated memory. [8].

Something to consider when allocating memory in the heap is that for a large program eventually all heap memory will be filled. In order to prevent this, memory needs to be freed when it is no longer being used. This can be implemented explicitly or implicitly. For explicit implementation, the programmer is in charge of handling deallocation while in an implicit implementation the garbage collection is handled automatically by the compiler. [3] We will focus on explicit garbage collection, by introducing a "del" keyword in the compiler. A problem with explicit deallocation to keep in mind is that freeing memory can be complex, and freeing something too soon by mistake leads to the program containing a *dangling pointer*. A dangling pointer points to memory that has been deallocated, and dereferencing such a pointer may have unpredicted effects. [3]

2.3 Classes and sub-classes

To increase the readability and the maintainability of programs that are written using ChocoPy, the language must provide some means that allow programmers to encapsulate variables and functions into objects. Therefore, classes and sub-classes are added to the ChocoPy language. A class is a template that describes variables and methods that exist within an object. Sub-classes, on the other hand, allow classes to inherit variables and methods from other classes [1]. Both classes and sub-classes are tools that help programmers to divide programs into many smaller entities. Those entities can be reused multiple times in the code, thus reducing the amount of duplicate code and increasing the maintainability of the program.

2.3.1 Memory

After creating an instance of a class, a block of memory on the heap is allocated for that class, and the base address of the class is returned to be saved in the variable. The allocated block size depends on the number of attributes the class has. The attributes are saved according to their order in the class definition. We can use the following formula to calculate the address of any attribute:

$$address = cba + (index - 1) * 8$$

Where **address** is the memory address of the attribute on the heap, **cba** is the base memory address of the class instance on the heap, and **index** is the index of the attribute in the class definition starting from one.
For example, the following code:

```
class Student :
    id: int
    age:Int
    def __init__(self:Student,id:int,age:int):
        self.id = id
        self.age = age
student1 = Student(1234,22)
student2 = Student(1332,23)
```

will be saved in this according to Figure 1 where every cell on the heap consists of 8 bytes

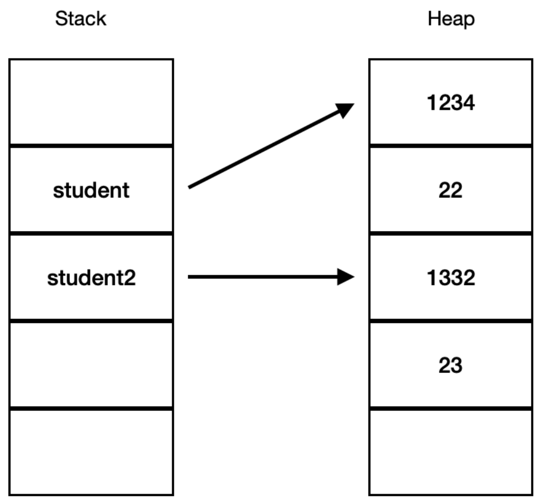

Figure 1. Class instance heap allocation

2.3.2 Method calls

Method calls are similar to other function calls that exist outside classes. However, every method must have access to the reference of the class that the method belongs to. This can be achieved by sending the reference of the class as an argument to the method call implicitly. For example, the method call will be rewritten by the compiler from:

```
shoppingCart = ShoppingCart()
shoppingCart.add(item)
```

to:

```
shoppingCart = ShoppingCart()
add(shoppingCart,item)
```

3 Heap allocation

In the code generation of the compiler, we begin all programs with allocating 5kb of memory on the heap using brk() as described in section 2.2. To keep track of where the next free location on the heap is we use a global buffer called heap_pointer, where we store and update its address. To first obtain the address of the start of the heap, brk() is called with 0 as argument. This returns the first available slot on the heap, and is what we initially store in the heap_pointer. Then we call on brk() again to allocate space on the heap. When we store data on the heap, each data slot has the length of 8 bytes.

3.1 The del function

We have implemented the key word del as a predefined function, but currently we do not have code generation for it and thus it does not do anything. Implementation of classes took more time than estimated, which made us run out of time. The del keyword can be used in future improvements for implementing manual memory management.

4 Implementation of lists

When it comes to our list implementation, we allocate one extra slot on the heap for each list. We decided that the first slot is the ’extra’ slot, and in it we store the size of the list. Storing the size this way makes iterating through a list straightforward. Due to the extra slot, we always need to add 1 to the index before retrieving or adding an element to a list. Furthermore, this method makes computing the length of a list very easy since you only need to retrieve the size stored in the first slot, which is the slot the list pointer points to. This is exactly how we implemented the len() function.

4.1 Static heap lists

These lists are fixed sized and allocated at compile time. One can allocate them using a similar syntax:

```
a:list = [1, 2, 3, 5]
```

4.2 Dynamic heap lists

These lists are fixed size lists that are allocated at run-time. They can be allocate them using a similar syntax:

```
a:list = [b] * n # line 1
c:list = [False, True, True] * 5 # line2
```

The code at line 1 will yield a list that has the element b repeated n times. This feature can be used to create lists of a certain size at run-time by multiplying the list with a number that is set by input. The list values can be changed at run time in the same way. The list multiplication syntax was chosen because Python does not have a specific syntax to create a list of a certain size like in java or C++.

This was implemented in the code generation by first multiplying the length of the given list with the number n. We store the new size in the first slot of the new array, which is located on the first available spot in the heap. Thereafter, we do a nested for loop in the assembly code which goes through the elements of the list n times, and adds them to the new list. This implementation allows choosing the size at run-time.

Since the compiler does not currently include strings, we cannot make informative prints to tell the user what input to give. Thus the type and order of input needed for a specific file needs to be described in the README file before a user runs a file.

5 Object oriented programming

For classes we ran in to some issues with existing name analysis methods. To get rid of these, large parts of the name analysis were refactored. We then added functionality for getting a class attribute by using the syntax a.b, where a is a class instance and b is an attribute. We did this by first introducing the syntax to the language and parser, and then implementing name analysis and type analysis for it.

5.1 Code generation

We altered the code generation for the class declaration and the object creation to change the objects’ allocations from dynamic stack allocations to dynamic heap allocations. The class declaration generates a constructor function for the class. This function instantiates all the class’s attributes on the heap and calls the ’init’ function if declared. After that, the function returns the heap memory address of the object. In addition to the constructor, all the methods of the class are generated after concatenating the name of the class is to the names of methods, i.e., this code

```
class Hello:
    def printer(self:Hello, a:int):
        print(a)
```

generates the following Assembly:

```
construct_Hello:
....
Hello_printer:
.....
```

When creating an object of a class, the constructor function is called and the memory address of the object is then saved into the variable. If a method is called from a class object, the name of the class is concatenated to the method's name, and then that function is called after passing the reference of that object to the method call, i.e., this code:

```
a:Hello = Hello()
a.printer()
```

generates the following Assembly:

```
call construct_Hello
....
call Hello_printer
.....
```

6 Evaluation

6.1 ChocoPy as a learning tool

Regarding the subject of ChocoPy being suitable for learning, we concluded that it seems to be appropriate. As [5] states, ChocoPy includes many features such as lists, strings and classes which all may serve to expand the students knowledge about compilers and the heap. This will be an advantage if the language is to be used in an advanced compilers course, where the students already know some things about implementing a compiler.

During the course of the implementation we ran in to many problems that were due to a rushed implementation of earlier parts of the compiler. Due to this, we strongly recommend that if the language were to be used in an advanced course, the assignments are either modular and independent from each other in the same way as [5], or that a optimized base program is provided for each assignment. This would help to avoid small errors in an early assignment make a later assignment near impossible without refactoring.

We also found that extending the language to implement the Mul list functionality was rewarding as it made us able to clearly see dynamic allocation on the heap.

6.2 Performance compared to Python3

An experiment was created an experiment to test the performance difference between the ChocoPy version that we implemented and Python3.

6.2.1 Experiment

The experiment was conducted by creating a script that constructs a list of 1500 numbers, then it uses the Bubble sort algorithm to sort the content of that list. The script was run 1500 times using a binary that was generated using the ChocoPy compiler and an additional 1500 times using Python3. After that all the execution times were saved to a file. Eventually, the execution times were used to calculate the average execution time, standard deviation, and the lower and the upper bounds of the confidence interval with the confidence level of 99.9 for both Python3 and the generated binary.

6.2.2 Results

Source	Mean	Std	L.Conf.Int.	U.Conf.Int.
ChocoPy	12.675	1.995	12.506	12.844
Python3	402.613	37.649	399.414	405.812

From the results table above, we can see that ChocoPy is much faster than Python3. The results can be explained with the paradigm difference between Python3 and ChocoPy. Python3 is an interpreted language where scripts gets interpreted by another program called the interpreter. On the other hand, ChocoPy is a compiled language where the compiler convert scripts into binary files that can be run directly by the machine and according to the Concepts of programming languages book[4], interpreted languages execution is 10 to 100 times slower than in compiled systems. Also Python is a dynamically typed language [4], which means that all the types are checked and determined at the run-time in contrast to ChocoPy where all the types are determined and checked at compile time and as a result, Python has to do more work during run-time which can increase the execution times of programs.

7 Related work

In [5], it is described that the intention behind the ChocoPy language which this report regards, is to suit for teaching about implementation and design of compilers. The chosen Python subset includes a lot of features that allows for a fairly high complexity in the implemented compilers, such as strings, classes and inheritance, lists of any type and method overloading to mention a few. The language has a detailed reference manual explaining its rules.

Using a subset of a well-known language is meant to help the students feel familiar with the rules, and stay motivated. The creators of ChocoPy further found a benefit of using a type-safe subset of such a dynamic language as Python to be that the students' compilers often were able to outperform the standard Python interpreter speed wise. They found that this helped to further improve the students' morale.

In our implementation we extended the ChocoPy language to also contain the Python list multiplication function. This allowed us to clearly see dynamic heap allocation. We believe that being able to see this important functionality, which is an important reason to use the heap in the first place, may have a similar motivating effect on the students as the one [5] found when letting the students compare their compilers to the Python interpreter, as it allows the student's to see that the heap allocation works accordingly.

In the trials of using ChocoPy in a compilers course, the students' work was divided into three different assignments. The assignments were modular in the way that they were independent of each other, so if a previous assignment had some minor faults it would not affect the later ones. Overall, they received very positive feedback on both the course and the usage of ChocoPy. [5]

In an earlier project [7] in the EDAN70 course, the suitability for ChocoPy when teaching about compilers was examined. At the end of the examination, the compiler had not been completely finished in time. It was however concluded that with the right adjustments, using a more controlled work structure, it should be possible to implement a compiler for ChocoPy and learn a lot from it within the given time frame. The resulting compiler was also compared to the standard Python interpreter and deemed to be faster, reaffirming what [5] had experienced during their trials.

In this report, we refactored and extended Tobias' [7] compiler to include more functionality. In our comparison experiment we similarly to both [5] and [7], got the result that our compiler was faster than the Python3 interpreter. We furthermore found the programming language to be suitable for more advanced compilers course due to the extendablity of the language, and this seems to be the experience of the authors of [5] as well, considering the positive feedback they received.

8 Conclusion

We successfully implemented heap-dynamic lists and altered the classes and class instances so that they are seemingly functional as well as stored on the heap. For memory management, we ran out of time and only implemented the keyword without any meaningful code generation.

Also, we were able to compare Python3 and ChocoPy performance wise and that ChocoPy is faster than Python3. While implementing lists and classes, we found ChocoPy to be suitable for learning more about compilers, as there were many features that had to be added. If ChocoPy is to be used in an advanced compilers course we suggest having modular assignments or that solid bases are provided for each assignment to prevent errors in earlier assignments from creating issues. Furthermore we believe it could possibly be beneficial to extend the ChocoPy language to include some dynamic heap allocation, as being able to showcase it may motivate the students the same way comparison to the Python interpreter has shown to do.

9 Future improvements

Due to running into more issues than estimated combined with a lack of time, we were not able to implement all the things we planned to. Here is a summary of some things that may be regarded as future improvements.

9.1 Memory management

Currently, heap allocation is only done at the beginning of the program which means that it may run out of space. To fix this, some type of memory management needs to be implemented i.e using the del keyword. An non-efficient quick-fix could also be to check if the heap_pointer will pass the heap_limit whenever an object is to be added to the heap. If it will pass, more space should be allocated to the heap before adding the object.

9.2 Inheritance

We did not have time to look into inheritance. This means that classes do not take inheritance into consideration currently, and thus need to be altered in order to handle them.

Acknowledgments

We want to thank Alfred Åkesson for his support and valuable input throughout the course of the project.

References

[1] Ray Klump. "Understanding object-oriented programming concepts". In: *2001 Power Engineering Society Summer Meeting. Conference Proceedings (Cat. No. 01CH37262)*. Vol. 2. IEEE. 2001, pp. 1070–1074.

[2] Torbjörn Ekman and Görel Hedin. "The JastAdd system — modular extensible compiler construction". In: *Science of Computer Programming* 69.1 (2007). Special issue on Experimental Software and Toolkits, pp. 14–26. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.02.003.

[3] Dick Grune et al. *Modern Compiler Design*. 2nd ed. New York Heidelberg Dordrecht London: Springer, 2012. ISBN: 978-1-4614-4698-9. DOI: 10.1007/978-1-4614-4698-9.

[4] R Sebesta. *Concepts of Programming Languages,Global Edition*. 11th ed. Pearson Education Limited, 2016. ISBN: 978-0-13-394302-3.

[5] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. "ChocoPy: A Programming Language for Compilers Courses". In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 41–45. ISBN: 9781450369893. DOI: 10.1145/3358711.3361627. URL: https://doi.org/10.1145/3358711.3361627.

[6] Berkeley University of California. *ChocoPy v2.2: RISC-V Implementation Guide*. Oct. 2019. URL: https://chocopy.org/chocopy_implementation_guide.pdf (visited on 12/01/2021).

[7] Tobias Karlsson. *ChocoPy compiler*. Tech. rep. LTH, 2020.

[8] Michael Haardt and Michael Kerrisk. *brk(2) - Linux manual page*. Mar. 2021. URL: https://man7.org/linux/man-pages/man2/brk.2.html (visited on 11/19/2021).