Contents

Chapter 1

Building and Gradualizing programming languages

Computers are complicated machines. A modern CPU (anno 2017) contains over *2 billion* transistors and flips states over *3 billion* times a second [1]. Controlling these machines is hard; controlling them with low-level assembly has been an impossible task for decades. Luckily, higher level programming languages have been created to ease this task.

However, creating such programming languages is a hard task too. Aside from the technical details of executing a language on a specific machine, languages should be formally correct and strive to minimize errors made by the human programmer, preferably without hindering creating usefull programs. This is a huge task; several approaches to solve this complex problem have been tried, all with their own trade-offs - such as usage of typecheckers, amongst other choices. Another hindrance is the lack of common jargon and tools supporting programming language design.

1.1 Representing arbitrary semantics

Program Language Design is a vast and intriguing field. As this field starts to mature, a common jargon is starting to emerge among researchers to formally pin down programming languages and concepts. This process was started by John Backus and Peter Naur in 1960, by introducing *BNF* in the famous ALGOL60 report [2], where the **syntax** of the ALGOL60 language was formally specified. Due to its simplicity and ease to use, it has become a standard tool for any language designer and has been used throughout of the field of computer science.

Sadly, no such formal language is availabe to reason about the **semantics** of a programming language. Researchers often use *natural deduction* to denote semantics, but in an informal way. We crystallize this by introducing a tool which allows the direct input of such rules -allowing manipulation directly on the parsetrees- giving rise to **parsetree oriented programming** and providing an intuitive interface to formally create programming languages, reason about them and execute them.

By explicitly stating the semantics of a programming language as formal rules, these rules can be automatically transformed and programming languages can be automatically changed.

In this master dissertation, we present a tool which:

- Allows an easy notation for both the syntax and semantics of arbitrary programming languages
- Which interprets these languages
- Provides ways to automatically reason about certain aspects and properties of the semantics
- And helps rewriting parts of the typesystem to gradualize them

The tool should help with easily creating and testing programming languages; it should help analyzing the various choices that can be made.

1.2 Static versus dynamic languages

One of those choices that programming languages make, is wheter a typechecker is used or not.

For example, consider the erronous expression `0.5 + True`.

A programming language with **static typing**, such as Java, will point out this error to developer, even before running the program. A dynamic programming language, such as Python, will happily start executing of the program, only crashing when it attempts to calculate the value.

This dynamic behaviour can cause bugs to go by undedected for a long time. For example, a bug is hidden in the following Python snippet. Can you spot it [1]?

```
1 if some_rare_condition:
2         list = list.sort()
3 x       = list[0]
```

Python will happily execute this snippet, until `some_rare_condition` is met and the bug is triggered - perhaps after months in production.

Java, on the other hand, will quickly surface this bug with a compiler error and even refuse to start the code altogether:

```
1 List<Integer> list = ...
2 if(someRareCondition){
3         // Error: Type mismatch: cannot convert from void to List
4         list = list.sort(intComparator);
5 }
6 int x   = list.get(0)
```

The typechecker thus gives a lot of guarantees about your code, even before running a single line of it. The strongest guarantee the typechecker gives is that code will not crash due to type errors. Furthermore, having precise type information gives other benefits, such as compiler optimizations, code suggestions, ease of refactoring, ...

However, this typechecker has a cost to the programmer. First, types should be stated explicitly and slows down programming. Second, some valid programs

[1] `list.sort()` will sort the list in memory and return `void`. `list = list.sort()` thus results in `list` being `void`. The correct code is either `list = list.sorted()` or `list.sort()` (without assignment).

can't be written anymore. While typechecking is a good tool in the long run to maintain larger programs, it is a burden when creating small programs.

Per result, programs often start their life as a small *proof of concept* in a dynamic language. When more features are requested, the program steadily grows beyond the point it can do without static typechecker - but when it's already to cumbersome and expensive to rewrite it in a statically typed language.

1.3 Gradual typesystems

However, static or dynamic typing shouldn't be a binary choice. By using a *gradual* type system, some parts of the code might be statically typed - giving all the guarantees and checks of a static programming language; while other parts can dynamically typed - giving more freedom and speed to development. A program where all type annotations are given will offer the same guarantees as a static language, a program without type annotations is just as free as a dynamic program. This means that the developer has the best of both worlds and can migrate the codebase either way as needed:

```
1  // Here we type statically
2  List<Integer> list = new ArrayList<>()
3  if(someRareCondition){
4          // Error: Type mismatch: cannot convert from void to List
5          list = list.sort(intComparator);
6  }
7
8  // Here, we work dynamically
9  ? x      = list.get(0)
10
11 x        = "Some string"
12 System.out.println(x + True)
```

Very little gradual programming languages exist - for an obvious reason: creating a gradual type system is a hard.

Gradual typing is a new research domain. It is not widly known nor well understood. Based on the paper of Ronald Garcia, **Abstracting Gradual Typing**, we attempt to *automate gradual typing* of arbitrary programming languages, based on the tool above.

Bibliography

[1] Intel. Intel core^{TM} i7-6950x processor extreme edition specifications. Technical report, Intel.

[2] J. W. Backus et al. Peter Naur. Report on the algorithmic language algol 60. *Communications of the Association for Computing Machinery*, 3, May 1960.