Contents

Chapter 1

Building and Gradualizing programming languages

Computers are complicated machines. A modern CPU (anno 2017) contains around *2 billion* transistors and switches states around *4 billion* times a second. Controlling these machines is hard; controlling them with low-level commands has been an impossible task for decades. Luckily, higher level programming languages have been created.

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.

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 these programming languages and concepts. This process was started by John Backus Naur in 1963, by introducing *BNF*, where the **syntax** of a language could be formally declared. 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. 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**, 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 rewrites parts of the typesystem to gradualize them

1.2 Static versus dynamic languages

One of those choices that programming languages make, is wehter 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 example, a bug is hidden in the following Python snippet:

```
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 = new ArrayList<>()
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 yout code, even before running a single line of code. The strongest guarantee it offers, is that the code will not crash due to type errors, as above. Also, having precise typing information, the compiler can optimize more.

However, this typechecker has a cost to the programmer. First, types should be stated explicitly. Second, some valid programs can't be written anymore. Thirdly, it slows down programming. While typechecking is a good tool in the long run to maintain larger programs, it is a burden when creating small programs.

Often, programs start their life as a small *proof of concept* in a dynamic language. When more features are requested, the program steadily grows beyond a point it would benefit from having a static checker - but becomes to expensive to rewrite in a statically typed language.

1.3 Gradual typesystems

However, static versus 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. 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)
```

1.3.1 Why don't more gradual programming languages exist?

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.

Chapter 2

Related Work and Goals

As *Programming Language Design* starts to take of as a major field within computer sciences, tools are starting to surface to formally define programming languages. But what should the ideal tool do, if we were to create it? And what does already exist?

2.1 Goals and nongoals

The ultimate goal should be to create a common language for language design, as this would improve the usage of formal design of languages. To gain widespread adoption, we should have as little barriers as possible, in installation, usage and documentation. It should be easy for a newcomer to use, without hindering the expressive power or available tools.

2.1.1 Tooling

Practical aspects are important - even the greatest tools lose users over unecassry barriers.

The first potential barrier is **installation** - which should be as smooth as possible. New users are easily scared by a difficult installation process, hindering adoption. Preferably, the tool should be availabe in the package repos. If not, installation should be as easy as downloading and running a single binary. Dependencies should be avoided, as these are often hard to deploy on the dev machine - as they might be hard to get, to install, having version conflicts with other tools on the machine, not being supported on the operating system of choice...

The second important feature is **documentation**. Documentation should be easy to find, and preferably be distrubeted alongside the binary.

Thirdly, we'll also want to be **cross-platform**. While most of the PL community uses a Unix-machine, we'll also should support widely used, non-free operating systems.

2.1.2 Embedded in another programming language?

Should we design the tool as library or domain specific language embedded in anohter programming language? Or should we create a totally new program-

ming language? Using an embedded language gives us a headstart, as we might use all of the builtin functionality and optimazations. The toll later on the road is high, however. Starting with a fresh language has quite some benifits:

First, the user does not have to deal with the host language at all. The user is forced to make the choice between learning the new programming language - which is quite an investment- or ignoring the native bits, and never having full control over it.

Related, by creating a fresh language, we can focus this language totally on what is needed. This means that it is easy to cut out any boilerplate, making the language more fun to use.

By not using a host language, we can also perform analysises on metafunctions. This is possible on small, well understood languages; something that is hard to do in a host language, where compilers span over 100'000 lines of code.

As last, we don't have to deal with installing a host compiler, skipping another dependency.

2.1.3

2.2 Yacc

Yacc (Yet Another Compiler Compiler), designed in 1970 was an early tool designed to automatically generate parsers from a given *BNF*-syntax. Depending on the parsed rule, a certain action could be specified - such as constructing a parse tree.

It was a major step to formally define the syntax of a programming language. The rest of the design is however left to the designer.

As this was the first widely available tool for this purpose, it has been tremendously popular, specifically within unix systems, albeit as reimplementation *GNU bison*. Implementations in other programming languages are widely available.

An example syntax in Yacc is:

```
1  %{
2  #include <stdio.h>
3  #include "y.tab.h"
4  %}
5
6
7  zone                    return ZONETOK;
8  file                    return FILETOK;
9  [a-zA-Z][a-zA-Z0-9]*    yylval=strdup(yytext); return WORD;
10 [a-zA-Z0-9\/.-]+         yylval=strdup(yytext); return FILENAME;
11 \"                      return QUOTE;
12 \{                      return OBRACE;
13 \}                      return EBRACE;
14 ;                       return SEMICOLON;
15 \n                      /* ignore EOL */;
16 [ \t]+                  /* ignore whitespace */;
```

2.3 ANTLR

ANTLR (ANother Tool for Language Recognition) is a more modern *parser generator*. This tool has been widely used as well, as it is compatible with many

programming languages, such as Java, C#, Javascript, Python, ...

Just as *YACC*, for each rule defined, a programmer specified action is performed; most often the construction of a parsetree.

ANTLR is used to parse the syntax of projects as Groovy, Jython, Hibernate, OpenJDK Compiler Grammer Project, Twitter's search query language, Cassandra and Processing.

An example grammer is:

```
1  options {
2          language="Cpp";
3  }
4
5  class MyExprParser extends Parser;
6
7  options {
8          k = 2;
9          exportVocab=MyExpr;
10         buildAST = true;
11 }
12
13
14 exprlist
15   : ( assignment_statement )* EOF!
16   ;
17
18 assignment_statement
19   : assignment SEMICOLON!
20   ;
21
22 assignment
23   : (IDENT ASSIGN )? expr
24   ;
25
26 primary_expr
27   : IDENT
28   | constant
29   | (LPAREN! expr RPAREN! )
30   ;
31
32 sign_expr
33   : (MINUS)? primary_expr
34   ;
35
36 mul_expr
37   : sign_expr (( TIMES | DIVIDE | MOD ) sign_expr)*
38   ;
39
40 expr
41   : mul_expr (( PLUS | MINUS ) mul_expr)*
42   ;
43
44 constant
45   : (ICON | CHCON)
46   ;
```

2.4 XText

XText is a modern tool to define grammers and associated tooling support. It is heavily tied-in into the Java Virtual Machine, as grammers are compiled to *Java*

Artifacts. https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html
XText focuses majorly on tooling support. Once a language is defined, an plugin for the *Eclipse IDE* can give code suggestions, syntax highlighting, hover information, ...

This is usefull for day to day programming, but not as usefull for formal language design.

It might be noted that *XText* uses *ANTLR* for parsetree generation.

```
grammar org.example.domainmodel.Domainmodel with
                        org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/model/Domainmodel"

Domainmodel :
        (elements+=Type)*;

Type:
        DataType | Entity;

DataType:
        'datatype' name=ID;

Entity:
        'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
        '}';

Feature:
        (many?='many')? name=ID ':' type=[Type];
```

2.5 LLVM

LLVM focusses mainly on the technical aspect of running programs on specific, real world machines. It contains an excellent intermediate *intermediate representation* of imperative programs, which can be optimized and compiled for all major computer architectures. LLVM is thus an excellent compiler backend.

As it focuses on the backend, *LLVM* is less suited for easily defining a programming language and thus for researching Language Design. As seen in their own tutorial, declaring a parser for a simple programming language takes *nearly 500 lines* of imperative C-code.

LLVM is not usefull as tool to design programming languages.

LLVM is thus a production tool, made to compile day-to-day programming languages in an efficient way. It would usefull to hook this as backend to *ALGT*, as to further automate the process of creating programming languages. This is however out of scope for this master dissertation.

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

2.6 PLT-Redex

PLT-Redex is a DSL implemented in Racket. It allows the declaration of a syntax as BNF and the definition of arbitrary relations, such as reduction or typing. *PLT-REdex* also features an automated checker, which generates random examples and tests arbitrary properties on them.

In other words, *PLT-redex* is another major step to formally define languages and was thus a major inspiration to ALGT.

As **PLT-Redex** is a DSL, it assumes knowledge of the host language, *Racket* . On one hand, it is easy to escape to the host language and use features otherwise not available. On the other hand, this is a practical barrier to aspiring Designers and hobbyists. A new language has to be learned -Racket is far from popular- and installed, which brings its own problems.

Furthermore, by allowing specification parts to be a full-fledged programming language, it hinders automatic reasoning about several aspects of the definition.

Thirdly, being a DSL brings syntax overhead of the host language. A fresh programming language, specifically for this task, allows to focus on a clean and to the point syntax.

```
1  #lang racket
2  (require redex)
3
4  (define-language L
5    (e (e e)
6       (λ (x t) e)
7       x
8       (amb e ...)
9       number
10      (+ e ...)
11      (if0 e e e)
12      (fix e))
13   (t (→ t t) num)
14   (x variable-not-otherwise-mentioned))
```

2.7 MAUDE

Maude System is a high-level programming language based on rewriting and equational logic. It allows a broad range of applications, in a logic-programming driven way. It might be used as a tool to get explore the semantics of programming, it does not meet our needs to easily define programming languages - notably because a lot of overhead is introduced in the tool, both cognitive and syntactic.

```
1  fmod NAT is
2    sort Nat .
3
4    op 0 : -> Nat [ctor] .
5    op s : Nat -> Nat [ctor] .
6  endfm
```

2.8 ALGT

ALGT, which we present in this dissertation, tries to be a generic *compiler front-end* for arbitrary languages. It should be easy to set up and use, for both hobbyists wanting to create a language and academic researchers trying to create a formally correct language.

ALGT should handle *all* aspects of Programming Language Design, which is the Syntax, the runtime semantics, the typechecker (if wanted) and the associated properties (such as *Progress* and *Preservation*) with automatic tests.

By defining runtime semantics, an interpreter is automatically defined and operational as well. This means that no additional effort has to be done to immediatly *run* a target program.

To maximize ease of use, a build consists of a single binary, containing all that is needed, including the tutorial and Manual.

ALGT is written entirely in Haskell. However, the user of ALGT does not have to leave the *ALGT*-language for any task, so no knowledge of Haskell is needed.

It can be easily extended with additional features. Some of these are already added, such as automatic syntax highlighting, rendering of parsetrees as HTML and LaTeX; but also more advanced features, such as calculation of which syntactic forms are applicable to certain rules or totality and liveability checks of meta functions.

2.9 Feature comparison

Feature	Yacc	ANTLR	XText	LLVM	PLT-Redex	Maude	AL
Syntax generation							
Generating parsers	✓	✓	✓		✓		✓
Left-recursion detection	✓						✓
Left-recursion handling	✓						
Runtime semantics							
Running target programs			✓		✓	✓	✓
Defining formal runtime semantics					✓	✓	✓
Tracing how evaluation happens					✓	✓	✓
Typechecked parsetree modification					✓		✓
Typechecker							
Defining a typechecker				✓	✓	✓	
Defining a formal typechecker					✓	✓	✓
Running a typechecker					✓	✓	✓
Automated testing			✓		✓		
Tooling							
Easy to set up and deploy	✓		✓	✓		✓	✓
Documentation	✓	✓	✓	✓		✓	✓
Automated tests					✓		✓
Automatic optimazation				✓			

Feature	Yacc	ANTLR	XText	LLVM	PLT-Redex	Maude	ALGT
Compilation				✓			
Syntax highlighting	✓				✓		
Editor support of target language			✓				