

Contents

1	Building and Gradualizing programming languages	3
1.1	Representing arbitrary semantics	3
1.2	Static versus dynamic languages	4
1.3	Gradual typesystems	5
2	Related Work and Goals	7
2.1	Goals and nongoals	7
2.1.1	Tooling	7
2.1.2	Metalinguage	8
2.1.3	Parsing the target language	8
2.1.4	Executing the target language	9
2.1.5	Typechecking the target language	9
2.2	Related Work	9
2.2.1	Yacc	9
2.2.2	ANTLR	10
2.2.3	XText	11
2.2.4	LLVM	11
2.3	PLT-Redex	12
2.4	MAUDE	12
2.5	ALGT	13
2.6	Feature comparison	13

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 available 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 whether a typechecker is used or not.

For example, consider the erroneous 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 undetected 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 too 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 be 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 // This is statically typed
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 // This is dynamic
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 widely 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 off as a major field within computer sciences, tools are surfacing 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 increase formalization of language design. To gain widespread adoption, there should be 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 used by the expert.

2.1.1 Tooling

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

The first potential barrier is **installation** - which should be as smooth as possible. New users are easily scared by a difficult installation process, fleeing to other tools hindering adoption. Preferably, the tool should be available 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 distributed alongside the binary.

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

As last, extra features like **syntax highlighting**, **automated tests** or having editor support for the target language is a nice touch.

2.1.2 Metalanguage

The most important part is the metalangue itself - as that is the major interface the language designer will use.

As language design itself is already difficult to grasp, we want our language to be **easy**. There are some aspects which help to achieve this.

An easy language should be:

- as **focused** as possible, with little boilerplate. The core concepts of language design should have a central place.
- **expressive** enough to be usefull
- as **simple** as possible, thus having as little elements and special constructs which should be learned and considered when doing automatic translations.
- **checked** as much as possible for big and small errors and report these errors with a clear error message.

Embedded in another programming language?

Should the tool be designed as library or domain specific language embedded in anohter programming language? Or should a totally new programming language be created?

Using an embedded language gives us a headstart, as all of the builtin functionality and optimazations can be used. However, The toll later on the road is high. Starting with a fresh language has quite some benefits:

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 a full grasp over the definition.

Second, by creating a fresh language, this language can be streamlined on what is needed - it is easy to cut out any boilerplate, making the language more fun to use.

By not using a host language, analyses on metafunctions become possible, as the metalanguage is small and well understood. This would be 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 Parsing the target language

The first step in defining our target language is declaring its syntax, for which the standard practice has been *BNF* since its introduction in the ALGOL-report. As BNF is the de facto standard, it is already well-known to language designers and thus both the theoritical and practical aspects are well understood. Furthermore, it is easy to port existing languages, as often a BNF is already available for this language.

A drawback is that many variants of BNF exist, each with their own superficial syntactic differences. This is only a minor drawback though: as the underlying structure is the same, a simple search-and-replace can easily convert one dialect into another. On the other hand, we want the BNF syntax to be as light and boilerplate-free as possible, eventually introducing a new dialect.

No other lower-level details should be exposed to the language designer, such as tokenizing or the internal definition of the parsetree - the language designer should be working directly with the parsetree.

2.1.4 Executing the target language

Of course, we'll want to execute our target programming language in one way or another. What are important aspects? As this is a tool to develop and prototype programming language, we're firstly concerned with debugging the programming language, no matter how the runtime semantics were declared.

- **Immediate feedback:** when starting the program, we want to see output as soon as possible
- **Write once, run anywhere:** running target programs should behave the same on all systems
- **traceable:** it should be possible to see how program execution went exactly, step by step, as to easily debug.

While compilation to a target architecture and target program optimizations are nice, they are not priorities - as it only complicates the implementation.

2.1.5 Typechecking the target language

Just like the operational semantics, we should be able to build and run a type-checker for the target language. Here, the same constraints apply. We should strive to make the declaration of the typesystem consistent with the operational semantics, preferably it should use the same kind of logic.

We should also strive to **automatically test** the correctness of the type-checker, in an automated way, in conjunction with the operational semantics.

2.2 Related Work

With these requirements in mind, we investigate what tools already exist and how these tools evolved.

2.2.1 Yacc

Yacc (Yet Another Compiler Compiler), designed in 1975 was the first 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, but carries a clear legacy of its inception era: you are supposed to include raw *c*-statements, compile to *c* and then compile the generated *c*-code. Furthermore, lexing and parsing are two different steps, requiring two different declarations. Thus, quite some low-level work is needed.

Furthermore, only the parser itself is generated, the parsetree itself should be designed by the language designer.

As this was the first widely available tool for this purpose, it has been tremendously popular, specifically within unix systems, albeit as reimplementations

GNU bison. Implementations in other programming languages are widely available. Yacc gets an honorable mention here for its historical importance, but is outdated.

2.2.2 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 Grammar Project, Twitter's search query language, Cassandra and Processing.

An example grammar 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   ;

```

2.2.3 XText

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

```

1 grammar org.example.domainmodel.Domainmodel with
2     org.eclipse.xtext.common.Terminals
3
4 generate domainmodel "http://www.example.org/model/Domainmodel"
5
6 Domainmodel :
7     (elements+=Type)*;
8
9 Type:
10     DataType | Entity;
11
12 DataType:
13     'datatype' name=ID;
14
15 Entity:
16     'entity' name=ID ('extends' superType=[Entity])? '{'
17     (features+=Feature)*
18     '}' ;
19
20 Feature:
21     (many?='many')? name=ID ':' type=[Type];

```

2.2.4 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.

```

1 | define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
2 |   entry:
3 |     %tmp = mul i32 %x, %y
4 |     %tmp2 = add i32 %tmp, %z
5 |     ret i32 %tmp2
6 | }

```

2.3 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))

```

2.4 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

```

2.5 *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.

2.6 Feature comparison

Bibliography

- [1] Intel. Intel coreTM i7-6950x processor extreme edition specifications. Technical report, Intel, 2017.
- [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.