Master Dissertation
Mathematical Computer Sciences
2016 - 2017

ALGT
A Framework to Describe and Gradualize Programming Languages

Pieter Vander Vennet

promoted by
Prof. Dr. Christophe Scholliers

GHENT
UNIVERSITY

Acknowledgements

For this master dissertation, I would like to express my gratitude to the following persons, without whom this master dissertation would not have been possible:

Christophe Scholliers, for offering interesting research topics, for requesting features I would have never thought of, for giving much needed advice on how to improve my writing skills, for making time in his busy life, for worrying about me when I was sick and generally for inspiring me.

The Faculty of Sciences, for offering a varied set of fascinating courses which I enjoyed;

Ilion Beyst, for always being enthousiastic about the entire tool, testing and abusing it in countless ways; and for always being up to date about the entire project, giving great advice;

The members of *Zeus WPI*, for their general interest and allowing me to host a tutorial evening, where some members tried to create their own programming language;

Isaura Claeys, for proofreading parts of the manual;

Rien Maertens, for being an excellent rubber duck, quacking the writers block away;

Tom Schrijvers, whom taught me to program in Haskell and let me discover the love for language design;

And my family, for supporting me throughout the journey, often exclaiming they had no idea what I was talking about or for testing the tool.

Contents

0.1 Related work

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

0.1.1 Yacc

Yacc (Yet Another Compiler Compiler), published in 1975 [**?**] was the first tool designed to automatically generate parsers from a given *BNF*-syntax. Depending on the parsed rule, a certain action can be specified - such as constructing a parsetree.

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. 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 had been tremendously popular, specifically within unix systems, albeit as reimplementation *GNU bison*. Implementations in other programming languages are widely available.

To modern standards, Yacc is outdated. Of the depicted goals, Yacc can only create a parser based on a given grammer, all the other work is still done by the language designer. Yacc gets an honorable mention for its historical importance, but is not relevant anymore today.

```
grammar STFL;

bool     : 'True'
         | 'False';

baseType        : 'Int'
                | 'Bool' ;

typeTerm        : baseType
                | '(' baseType ')';

type     : typeTerm '->' type
         | typeTerm
         ;

ID       : [a-z]+ ;

INT      : '0'..'9'+;

value    : bool
         | INT;

e        : eL '+' e
         | eL '::' type
         | eL e
         | eL ;

eL       : value
         | ID
         | '(' '\\' ID ':' type '.' e ')'
         | 'If' e 'Then' e 'Else' e
         | '(' e ')';

WS : [\t\r\n ]+ -> skip;
```

Figure 1: The grammar of STFL in ANTLR

0.1.2 ANTLR

ANTLR (ANother Tool for Language Recognition) is a more modern *parser generator* [**?**], created in 1989. This tool has been widely used as well, as it is compatible with many programming languages, such as Java, C#, Javascript, Python, ... With this grammer, a parsetree for the input is constructed. Eventually, extra actions can be performed for each part of the parsetree while parsing.

Apart from this first step, the rest of the language design is left to the programmer, which has to work with a chosen host language to build the further steps. ANTRL has the fundamental different goal to create an efficient parser in a language of choice as start of another toolchain.

As example, an implementation for STFL is given in figure 1

0.1.3 XText

XText [**?**] is a modern tool to define grammers and associated tooling support. The main use and focus of XText is providing the syntax highlighting, semantic autocompletion, code browsing and other tools featured by the Eclipse IDE. It is thus heavily integrated with java and the Eclipse ecosystem, thus a working knowledge of Java and the Eclipse ecosystem is required

- even requiring a working installation of both.

Parsing grammers is done with a metasyntax heavily inspired by ANTLR, enhanced with naming entities and cross references.

In conclusion, XText is a practical tool supporting IDE features, but clearly not suited for the language prototyping we intend to do.

0.1.4 LLVM

LLVM [?] focusses on the technical aspect of running programs as fast as possible on specific, real world machines. Working with an excellent *intermediate representation* of imperative programs, LLVM optimizes and compiles target programs to all major computer architectures.

As it focuses on the compiler 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 an industrial strength production tool, made to compile everyday programming languages in an efficient way. While it is an extremely usefull piece of software, it's goals are the exact opposite of what we want to achieve.

It would usefull to hook *LLVM* as backend to a language prototyping tool to further automate the process of creating programming languages. This is however out of scope for this master dissertation.

0.1.5 MAUDE

Maude System [?] [?] is a high-level programming language for 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 explore the semantics of programming, but it does not meet our needs to easily define programming languages - notably because overhead is introduced in the tool, both cognitive and syntactic to define even basic languages.

While rewriting rules play a major role in defining semantics, Maude serves as vehicle to experiment with logic and the basics of computation and is less geared toward programming language development.

0.1.6 PLT-Redex

PLT-Redex [?] is a DSL implemented in Racket, allowing 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.

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 new designers and hobbyists. A new user has to learn a new language, including all the aspects not optimized for language design.

In other words, *PLT-redex* is another major step to formally and easily define languages and was a major inspiration to ALGT. However, the approach to embed it within Racket hinders adoption for unexperienced users and blocks automatic reasoning on metafunctions - at least for someone who does not want to dive into Racket and the internals of a big project.

An example can be seen in figure 2.

```
#lang racket
(require redex)
(define-language L
   ( e (e e)
       (λ (x t) e)
       x
       (amb e ...)
       number
       (+ e ...)
       (if0 e e e)
       (fix e))
    (t (→ t t) num)
    (x variable-not-otherwise-mentioned))

(define-metafunction L+Γ
  [(different x_1 x_1) #f]
  [(different x_1 x_2) #t])

(define-extended-language Ev L+Γ
  (p (e ...))
  (P (e ... E e ...))
  (E (v E)
     (E e)
     (+ v ... E e ...)
     (if0 E e e)
     (fix E)
     hole)
  (v (λ (x t) e)
     (fix v)
     number))
```

```
(define-metafunction Ev
  Σ : number ... -> number
  [(Σ number ...), (apply + (term (number ...)))])
(require redex/tut-subst)
(define-metafunction Ev
  subst : x v e -> e
  [(subst x v e) ,(subst/proc x? (list (term x)) (list (term v)) (term e))])
(define x? (redex-match Ev x))
(define red
  (reduction-relation
   Ev
   #:domain p
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 v e_1 e_2))
        (in-hole P e_2)
        (side-condition (not (equal? 0 (term v))))
        "if0f")
   (--> (in-hole P ((fix (λ (x t) e)) v))
        (in-hole P (((λ (x t) e) (fix (λ (x t) e))) v))
        "fix")
   (--> (in-hole P ((λ (x t) e) v))
        (in-hole P (subst x v e))
        "βv")
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (e_1 ... (in-hole E (amb e_2 ...)) e_3 ...)
        (e_1 ... (in-hole E e_2) ... e_3 ...)
        "amb")))
```

Figure 2: Grammar definition and reduction rules for STFL in PLT-redex, compiled from the tutorial at [?]

0.1.7 OTT

OTT [?] is another major step in the automate formalization of langauges. An OTT-langage is defined in its own format, after which it is translated to either LaTeX for typesetting or Coq, HOL, Isabelle or Twelf.

Translating the OTT-metalanguage into a proof assistent language has the drawback that knowing such a language is thus a requirement for using OTT. The philosophy of the tool seems to be helping language designers of whom the main tool already is such a language.

By using a new language, the syntax of OTT is more streamlined. However, as the tool translates into a proof assistent language, it still has to make some compromises, such as declaring the datatype a metavariable has.

In conclusion, OTT is another major step to formalization, but has high hurdles for new users. However, both the syntax and concepts of OTT have been an important inspiration.

As example, the grammer definition for STFL is given in figure 3.

0.1.8 The Gradualizer

The Gradualizer [?] is a research tool designed specifically to create gradual programming languages. No documentation exists at all, neither in the source code or on usage. The input and output format are written in λ-Prolog, which is not a widely used language and certainly not suitable for a beginner. The goal of the Gradualizer is to do research specifically on gradualizing certain typesystems automatically and is thus a specialized tools which only the experts know how to operate.

The gradualizer can handle some typesystems fully automaticly, at the cost of limiting the typesystems that can be gradualized.

An example implementation of STFL can be found in figure 4.

0.1.9 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.

An example of ALGT can be found in figure 5 and 6. This language will be explained in more detail in chapter **??**.

```
metavar termvar, x ::=
  {{ isa string }} {{ coq nat }} {{ coq-equality }} {{ hol string }}
  {{ ocaml int }} {{ tex \mathit{[[termvar]]} }} {{ com  term variable  }}

metavar typvar, X ::=
  {{ isa string }} {{ coq nat }} {{ coq-equality }} {{ hol string }}
  {{ ocaml int }} {{ tex \mathit{[[typvar]]} }} {{ com  type variable  }}

grammar
  t :: 't_' ::=                                         {{ com term }}
    | x                   ::   :: Var                     {{ com variable }}
    | \ x . t             ::   :: Lam  (+ bind x in t +)  {{ com abstraction }}
    | t t'                ::   :: App                     {{ com application }}
    | ( t )               :: S :: paren   {{ ich [[t]] }} {{ ocaml int }}
    | { t / x } t'        :: M :: tsub    {{ ich ( tsubst_t [[t]] [[x]] [[t']] ) }} {{ ocaml int }}

  v :: 'v_' ::=                                         {{ com  value }}
    | \ x . t             ::   :: Lam                     {{ com abstraction }}

  T :: T_ ::=                                           {{ com type }}
    | X                   ::   :: var                     {{ com variable }}
    | T -> T'             ::   :: arrow                   {{ com function }}
    | ( T )               :: S :: paren {{ ich [[T]] }} {{ ocaml int }}

  G {{ tex \Gamma }} :: G_ ::= {{ isa (termvar*T) list }} {{ coq list (termvar*T) }} {{ ocaml (termvar*T) list }}
                               {{ hol (termvar#T) list }} {{ com type environment }}
    | empty               ::   :: em
        {{ isa Nil }}
        {{ coq G_nil }}
        {{ hol [] }}
    | G , x : T           ::   :: vn
        {{ isa ([[x]],[[T]])#[[G]] }}
        {{ coq (cons ([[x]],[[T]]) [[G]]) }}
        {{ hol (([[x]],[[T]])::[[G]]) }}

  terminals :: 'terminals_' ::=
    | \                   ::   :: lambda     {{ tex \lambda }}
    | -->                 ::   :: red        {{ tex \longrightarrow }}
    |  ->                 ::   :: arrow      {{ tex \rightarrow }}
    | |-                  ::   :: turnstile  {{ tex \vdash }}
    | in                  ::   :: in         {{ tex \in }}
```

Figure 3: The grammar definition of a simply typed calculus, declared in OTT. This definition can be downloaded freely from the OTT-site [?]

```
sig STFL_if_int


kind    term                    type.
kind    typ                             type.

type    int                     typ.
type    bool                    typ.
type    arrow           typ -> typ -> typ.

type    app                     term -> term -> term.
type    abs                     typ -> (term -> term) -> term.

type    typeOf                  term -> typ -> o.

type    add             term -> term -> term.
type    zero            term.
type    succ            term -> term.

type            if              term -> term -> term -> term.
type            tt      term.
type            ff      term.

% contravariant arrow 1.

module STFL_if_int.

typeOf (abs T1 E) (arrow T1 T2) :- (pi x\ (typeOf x T1 => typeOf (E x) T2)).
typeOf (app E1 E2) T2 :- typeOf E1 (arrow T1 T2), typeOf E2 T1.
typeOf (add E1 E2) (int) :- typeOf E1 (int), typeOf E2 (int).
typeOf (zero) (int).
typeOf (succ E) (int) :- typeOf E (int).
typeOf (if E1 E2 E3) T :- typeOf E1 (bool), typeOf E2 T, typeOf E3 T.
typeOf (tt) (bool).
typeOf (ff) (bool).
```

Figure 4: The grammer definition and reduction of a simply typed calculus, declared in λ-Prolog. This example can found on the website showcasing the gradualizer [**?**]

```
 STFL
******

# A Simply Typed Function Language

 Syntax
========

bool     ::= "True" | "False"



baseType::= "Int" | "Bool"
typeTerm::= baseType | "(" type ")"
type     ::= typeTerm "->" type | typeTerm



var      ::= Identifier
number   ::= Number
value    ::= bool | number

canon    ::= value | "(" "\\" var ":" type "." e ")"
e        ::= eL "+" e
         | eL "::" type
         | eL e
         | eL

eL       ::= canon
         | var
         | "If" e "Then" e "Else" e
         | "(" e ")"
```

Figure 5: The grammer definition of ALGT

```
 Functions
===========

dom                         : type -> typeTerm
dom("(" T1 ")")             = T1
dom(("(" T1 ")") "->" T2)   = T1
dom(T1 "->" T2)             = T1

cod                         : type -> type
cod("(" T2 ")")             = T2
cod(T1 "->" ("(" T2 ")"))   = T2
cod(T1 "->" T2)             = T2

 Relations
===========

(→)      : e (in), e (out)        Pronounced as "smallstep"

 Rules
=======

 e0 → e1
-----------------                        [EvalCtx]
 e[e0] → e[e1]

 n1:Number      n2:Number
------------------------------------     [EvalPlus]
 n1 "+" n2 → !plus(n1, n2)

 e :: T0        T == T0
------------------------                 [EvalAscr]
 e "::" T → e


----------------                         [EvalParens]
 "(" e ")" → e


---------------------------------------          [EvalIfTrue]
 "If" "True" "Then" e1 "Else" e2 → e1


----------------------------------------         [EvalIfFalse]
 "If" "False" "Then" e1 "Else" e2 → e2


 arg:value      arg :: T
------------------------------------------------ [EvalLamApp]
 ("(" "\\" var ":" T "." e ")") arg → !subs:e(var, arg, e)
```

Figure 6: The reduction rule (with helper functions) of STFL in ALGT

0.2 Feature comparison

In the following table, a comparison of the related tools are provided.

The **metalanguage** is the language which handles the next steps, such as declaring a reduction rule or a typechecker.

- The metalanguage is *focused* if parsetrees can be modified with little or no boilerplate.
- The metalanguage is *simple* if it requires little extra knowlegde, e.g. knowlegde of a host programming language.
- If type-errors are detected in the metalanguage, then it is *typechecked*. This is important, as it prevents construction of malformed parsetrees and other easily preventable errors.

The second aspect is the **parsing** of the target language, which takes into account:

- *BNF-oriented* implies that the tool constructs a parser based on a context free grammer, described in BNF or equivalent format.
- *Light-syntax* indicates if the grammer syntax contains little boilerplate and without extra annotations which might confuse an unexperienced reader.
- *Parsetree-abstraction* indicates that the user never has to create a datatype for the parsetree and that the parsetree is automatically constructed.

To **execute** the programming language, following aspects are considered:

- *Correctness* implies that the metalanguage tries to ease reasoning about the semantics
- *Immediate feedback* means that as much usefull feedback is given as soon as possible, such as warnings for possible errors
- *Cross-platform* execution of the target language is possible, e.g. by having an interpreter available on all major platforms.
- *Debug information or traces* are usefull when prototyping a new language, to gain insights in how exactly a program execution went.

Constructing the **typechecker** should be considered too:

- Preferably, the typechecker can be constructed using the same metaconstructions as the interpreter
- Automatic and/or randomized tests should be performed
- The tools should gradualize or help gradualizing the typesystem

At last, **tooling** is explored, which describes the practical ease of use. This takes into account:

- How long *installation* took and how easy it was.
- If good *documentation* is easily available.
- If the program runs on *multiple platforms* (only tested on Linux).
- Whether a *syntax higlighter* exists for the created language.
- An option to render the typesystem as LaTeXresults in a checkmark for *typesetting*.

Feature	ANTLR	XTEXT	Maude	PLT-Redex	OTT	The Gradualizer	ALGT
Metalanguage							
Focused			✓	✓	✓		✓
Simple			✓	✓			✓
Typechecked			✓		✓		✓
Parsing							
BNF-oriented	✓	✓		✓	✓		✓
Light syntax	✓						✓
Parsetree-abstraction	✓	✓		✓	✓		✓
Execution							
Correctness			✓	✓	✓		✓
Immediate feedback			✓	✓	✓		✓
Cross-platform				✓			✓
Debug information/Traces			✓	✓			✓
Typechecker							
Constructed similar to semantics				✓	✓		✓
Automatic tests				✓			✓
Gradualization						✓	✓
Tooling							
Easy installation	✓	~	✓	✓	~		✓
Documentation	✓	✓	✓	✓	✓		✓
Cross-platform	✓	✓	~	✓	~	✓	✓
Syntax Highlighting		✓					✓
Latex Typesetting				✓	✓	✓	

0.3 Conclusion

While all tools are usefull, no tool fills the needs of our dissertation. Some of the tools, such as Yacc and Antlr, only focus on the actual parsing of the target language. Some other tools, such as XText or LLVM fill in other needs, such as tooling support or low-level code compilation.

PLT-redex is an excellent tool for formal language design, altough it lacks support for gradual languages. On the other hand, the Gradualizer helps gradualizing languages, but is hard to use and restricted to small languages.

No tool allows both easy design of programming languages and gradualization of them. In this disseration, a new tool named ALGT is introduced which allows both.