Contents

1 Building and Gradualizing Programming Languages **5**
- 1.1 Representing arbitrary programming languages . . . 5
  - 1.1.1 Describing syntax . . . 5
  - 1.1.2 Describing semantics . . . 6
  - 1.1.3 Mechanized checking of syntax and semantics . . . 7
- 1.2 Static versus dynamic languages . . . 8
- 1.3 Gradual typesystems . . . 9
- 1.4 Main contribution . . . 9

2 Related Work and Goals **11**
- 2.1 Goals and nongoals . . . 11
  - 2.1.1 Metalanguage . . . 11
  - 2.1.2 Parsing the target language . . . 12
  - 2.1.3 Executing the target language . . . 12
  - 2.1.4 Typechecking the target language . . . 13
  - 2.1.5 Tooling . . . 13
- 2.2 Related work . . . 13
  - 2.2.1 Yacc . . . 13
  - 2.2.2 ANTLR . . . 14
  - 2.2.3 XText . . . 15
  - 2.2.4 LLVM . . . 15
  - 2.2.5 MAUDE . . . 15
  - 2.2.6 PLT-Redex . . . 15
  - 2.2.7 OTT . . . 16
  - 2.2.8 The Gradualizer . . . 16
  - 2.2.9 ALGT . . . 16
- 2.3 Feature comparison . . . 22
- 2.4 Conclusion . . . 24

3 ALGT in a Nutshell **25**
- 3.1 STFL . . . 25
- 3.2 Skeleton . . . 26
- 3.3 Syntax . . . 26
  - 3.3.1 BNF . . . 26
  - 3.3.2 STFL-syntax . . . 27
  - 3.3.3 Parsing . . . 28
  - 3.3.4 Conclusion . . . 29

3.4 Metafunctions . . . 30
3.4.1 Domain and codomain . . . 31
3.4.2 Equate . . . 33
3.4.3 Safety checks . . . 33
3.4.4 Conclusion . . . 34
3.5 Natural deduction . . . 35
3.5.1 Giving meaning to a program . . . 35
3.5.2 Semantics as relations . . . 35
3.5.3 Declaring relations . . . 35
3.5.4 Natural deduction rules . . . 36
3.5.5 Defining smallstep . . . 40
3.5.6 Typechecker . . . 44
3.5.7 Declaration of :: and $\vdash$ . . . 44
3.5.8 Definition of $\vdash$ . . . 45
3.5.9 Definition of :: . . . 47
3.5.10 Overview . . . 48
3.5.11 Conclusion . . . 49
3.6 Declaring and checking properties . . . 49
3.6.1 Progress and Preservation . . . 49
3.6.2 Stating Preservation . . . 49
3.6.3 Stating Progress . . . 49
3.6.4 Testing properties . . . 50
3.7 Conclusion . . . 51

4 Syntax Driven Abstract Interpretation 53
4.1 Abstract interpretation . . . 53
4.1.1 The rule of signs . . . 54
4.1.2 Ranges as abstract domain . . . 56
4.1.3 Collecting semantics . . . 56
4.1.4 Properties of α and γ . . . 57
4.2 Properties of the syntax . . . 59
4.2.1 Syntactic forms as sets . . . 59
4.2.2 Embedded syntactic forms . . . 59
4.2.3 Empty sets . . . 59
4.2.4 Left recursive grammers . . . 61
4.2.5 Uniqueness of sequences . . . 63
4.3 Representing sets of values . . . 64
4.3.1 Sets with concrete values . . . 64
4.3.2 Symbolic sets . . . 64
4.3.3 Infinite sets . . . 65
4.3.4 Set representation of a syntax . . . 65
4.3.5 Conclusion . . . 65
4.4 Operations on representations . . . 66
4.4.1 Addition of sets . . . 66
4.4.2 Unfolding . . . 66
4.4.3 Refolding . . . 67
4.4.4 Resolving to a syntactic form . . . 70
4.4.5 Subtraction of sets . . . 70
4.4.6 Conclusion . . . 72

4.5 Lifting metafunctions to work over sets . . . 72
4.5.1 Translating metafunctions to the abstract domain . . . 72
4.5.2 Calculating a possible pattern match . . . 73
4.5.3 Calculating possible return values of an expression . . . 76
4.5.4 Combining clauses . . . 76
4.6 Conclusion . . . 77

5 Gradualization of STFL **79**
5.1 Gradual languages . . . 79
5.1.1 Some gradualization of STFL . . . 80
5.2 Breaking preservation . . . 81
5.3 Abstracting Gradual Typing . . . 82
5.3.1 The proposed dynamic runtime . . . 82
5.3.2 Simplifying the runtime . . . 82
5.4 Gradualizing the typesystem manually . . . 83
5.4.1 Determining the scope of the unknown type information . . . 83
5.4.2 Representing dynamic types . . . 83
5.4.3 Gradualizing the typing relationship . . . 84
5.5 The dynamic runtime . . . 86
5.6 Automating gradualization . . . 88
5.6.1 Refactoring . . . 88
5.6.2 Gradualizing domain and codomain . . . 89
5.6.3 Gradualizing equate . . . 90
5.6.4 Clause additions . . . 92
5.7 Conclusion . . . 92

6 Implementation of ALGT **93**
6.1 Representation and parsing of arbitrary syntax . . . 93
6.1.1 Target language parsing . . . 94
6.2 Target program representation . . . 94
6.3 Metafunctions . . . 94
6.3.1 Typechecker of expressions . . . 97
6.4 Interpretation of metafunctions . . . 102
6.4.1 Pattern matching . . . 102
6.4.2 Parsetree construction . . . 102
6.5 Relations . . . 105
6.5.1 Typechecking . . . 105
6.5.2 Building proof of a relation . . . 105
6.5.3 Proving a relation with evaluation contexts . . . 107
6.6 Further reading . . . 107

7 ALGT as tool for Language Design **109**
7.1 Context . . . 109
7.1.1 Related work . . . 109
7.2 Main contribution . . . 110
7.2.1 Tool for language design . . . 110
7.2.2 Tool for gradualization . . . 111
7.3 Conclusion . . . 112

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 [**?**]. 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 aspect of executing a program in such 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 the usage of a type-checkers, amongst other choices. Another hindrance for the programming language field is the lack of common jargon and tools supporting programming language design.

1.1 Representing arbitrary programming languages

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. In this section, a short overview is given of what techniques are in use today and how these evolved historically.

1.1.1 Describing syntax

The process of formally describing programming languages has its root in linguistics, the scientific study of natural languages. Linguistics has been a research topic for millenia: the first known linguist, Pāṇini lived between the 6th and 4th century BCE in India [**?**]. He described Sanskrit using a notation with equivalent power as techniques in use today. [**?**]

The first Western linguist whom studied language using the underlying block structure was Noam Chomsky. Chomsky tried, amongst other approaches, to create a generative grammar for English. These generative grammars would later be named _context free grammars. Sadly, Chomsky concluded that such grammars are not powerfull enough to capture all the complexities of natural languages [**?**] such as English.

Despite this failure, context free grammars are all but useless: they are an excellent vehicle to describe the syntax of programming languages. John Backus created a straightforward way to describe context free grammars, which he formalized together with Peter Naur as Backus-Naur-Form or BNF. BNF was introduced to the world in 1960, when the syntax of ALGOL60 was specified formally using BNF in the famous ALGOL60 report [**?**]. Due to its simplicity and ease of use, BNF has become a standard tool for any language designer and has been used throughout the field of computer science.

1.1.2 Describing semantics

All programs achieve an effect when run. This effect is created by some sort of transformation of the world, such as the manipulation of files, screen output, activation of a motor, playing sound, ... The calculation of an expression is an effect as well, as the human interpreting the result will act accordingly.

This effect is what gives meaning to a program. The meaning of any program, thus of the language as a whole, is called the **semantics** of that language.

Describing this effect is not as straightforward as describing the syntax. Since the end of 1960, various techniques and frameworks have been proposed to describe semantics.

- The first, rather informal way to describe semantics is by **translating** a language into another another language. This translation is not practical when carried out by humans, but tremendously usefull when done by a computer as this takes away the need to work directly with machine code. The first such system, *Speedcode* by John Backus [**?**], translated higher-level statements and executed them on IBM 701 machines.
- The first formal, mathematical framework to describe semantics was introduced in 1968 by the ALGOL68-report [**?**]. ALGOL68 was the first language to have a full formal definition of the semantics using the mathematical framework that is known today as **operational semantics**.
- Another mathematical framework to describe semantics was introduced a year later by C.A.R. Hoare [**?**], in which he introduces an **axiomatic framework** to describe programming languages.
- The last approach capturing semantics still widely used today was introduced in 1971 by Dana Scott and Christopher Strachey. They introduce the technique to translate the program to a mathematical object (such as a function, transforming the input to the output), giving rise to **denotational semantics** [**?**].

Of course, all these techniques have their own benefits and drawbacks.

Translation to another language

The first way to let a target program transform the world, is by translating the program from the target language to a host language (such as machine code, assembly, C, ...). Afterwards, this translated program can be executed on a real-life machine. Such translation is necessary to create programs executing as fast as possible on real hardware. However, proving a property about the target program becomes cumbersome:

- First, the host language should be modelled and a translation from target to host language defined
- Then, an analogous property of the host language should be proven
- Third, it should be proven that the translation preserves the property.

Although this is possible, this dissertation is focused on theoretical properties and automatic transformation of a programming language. These would become needlessly complicated using this technique, thus it is not considered here.

Denotational semantics

The second way to give meaning to a program, are denotational semantics. Denotational semantics try to give a target program meaning by using a *mathematical object* representing the program. Such a mathematical object could be a function, which behaves (in the mathematical world) as the target program behaves on real data.

The main problem with this approach is that a *mathematical object* is, by its very nature, intangable and can only described by some *syntactic notation*. Trying to capture mathematics result in the creation of a new formal language (such as FunMath), thus only moving the problem of giving semantics to the new language. Furthermore, it still has all the issues of translation as mentioned above.

Axiomatic semantics

Axiomatic semantics, introduced by C.A.R. Hoare [**?**], do not give the state changes explicitly, but rather describe the changes on assumed properties of the state. This often happens in the form of Hoare triples, which consist of the preconditions, program and postconditions - the conditions which should be met before the program execution, so that the state after program execution yields the given postconditions.

This moves the problem of semantics from the target language onto a host language, apart from the need to proof properties of the host language.

Structural operational semantics

Structural operational semantics tries to give meaning to the entire target program by giving meaning to each of the parts [**?**]. This can be done in an inherently syntactic way, thus without leaving ALGT.

- Expressions without state (such as `5 + 6`) can be evaluated by replacing them with their result (`11`).
- Imperative programs are given meaning by modelling a state. This state is a syntacic form which acts as data structure, possibly containing a variable store, the output, a file system... Each statement of the imperative language has an effect on this state, which can now be formalized too.

Operational semantics is the most straightforward approach for this dissertation, as will be noted in section 3.5. Using this framework, semantics for a simple programming language will be constructed in section 3.5.5.

1.1.3 Mechanized checking of syntax and semantics

Researchers often use BNF to specify the syntax and one of the above frameworks to denote semantics, but in an informal way: the semantics are denoted in LaTeX for publication. This is error-prone, as no mechanical checks are performed on their work.

This habit is changing lately as programming language researchers are starting to use various tools to automate this task. There are two broad categories of those tools:

- On one hand, theorem provers are used to automate the reasoning about the semantics: tools such as COQ and Isabelle are gaining popularity in the field, checking correctness proofs of the languages, but are quite complicated to use.
- On the other hand are lightweight tools used, optimized for language design. These tools help with typesetting, translation to the theorem provers or other smaller tasks, in a simple and easy metalanguage.

A review of these tools can be found in section 2.2.

1.2 Static versus dynamic languages

A tradeoff all programming languages make is the choice between static and dynamic typing, thus whether type errors are searched and detected before running the program or cause crashes during execution.

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

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

This dynamic behaviour can cause bugs to go undedected for a long time, such as the bug 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 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 can't be written anymore. While typechecking is a good tool in the long run to maintain large 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] `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).

1.3 Gradual typesystems

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  // 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
12 x        = "Some string"
13 System.out.println(x + True)
```

Very little gradual programming languages exist, because creating a gradual type system is a hard.

While gradual typing has been a research topic for decades [**?**], it is not widely known nor well understood. Based on the recent paper of Ronald Garcia, **Abstracting Gradual Typing** [**?**], we attempt to *automate gradual typing* of arbitrary programming languages, based on the tool above.

1.4 Main contribution

In this master dissertation, a new tool is presented which allows the design of arbitrary programming languages and helps gradualizing them. This tool, ALGT, is a new programming language designed for the rapid development of (gradual) programming languages.

Specifying programming languages is done using a lightweight denotation, capturing both the syntax and semantics in a formally correct way. This **new metalanguage for programming languages** is optimized for ease of use, minimizing boiler plate and unneeded elements. In chapter 3, a small functional language is specified using this metalanguage.

Using a given specification, the tool can **automatically build an interpreter** for this language: The grammar is used to construct a parser for the target program whereas the semantics are used to interpret said program. Readers interested in the machinery to achieve this, can find the algorithms to achieve this in chapter 6.

Based on the syntax of a language, an algorithm for **syntax driven abstract interpretation** is presented in chapter 4. This framework allows the analysis of metafunctions over a set of values in one pass. Apart from enabling some extra checks of the specification, it is a major step to automatic gradualization.

Automated gradualization is demonstrated in chapter 5, where the earlier introduced functional language is transformed into its gradual counterpart. For easily transforming the

language, a *language-changes* format is introduced which describes the differences between two dialects. The construction this file is aided by the abstract interpretation framework.

At last, the metalanguage, tool and gradualization are evaluated in chapter 7.

Chapter 2

Related Work and Goals

As *Programming Language Design* starts to take of as a major field within computer sciences, tools are surfacing which formally define programming languages. All of these tools are created within their own timeframe and goals in mind. In this chapter, we explore what aspects are important for language designers, thus giving the goals for such a tool. Then, we investigate what tools already exist in this space.

2.1 Goals and nongoals

The ultimate goal is to create a common language for language design as this increases the 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 Metalanguage

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

As language design itself is already difficult to grasp, we want the metalanguage to be **simple and 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 little elements and special constructs. Each construction should be learned by the user; and each construction should be accounted for when doing automatic reasoning or transformations of the language.
- **Checked** as much as possible for big and small errors and report these errors with a clear error message.

Embedded DSL versus a new standalone language

Such a tool can be implemented as library (or domain specific language) embedded in anohter programming language. The other option is creating a totally new programming language, with a standalone interpreter or compiler.

Implementing the tool as library in a host language gives us a headstart, as all of the builtin functionality and optimazations can be used. However, the cost later on is high. Starting with a fresh language has quite some benefits:

- The user does not have to deal with the host language at all. As embedded library, the language designer is forced either learning the new programming language (which takes a lot of time) or ignoring the native bits, and never having a full grasp of his creation.
- By creating a fresh language, its syntax can be streamlined on what is needed: boilerplate can be avoided, making the language more fun to use.
- By not using a host language, analysises on metafunctions become possible. Because the metalanguage is small, well understood and explicitly represented in a data structure, it can be modified. This would be hard using a modern host language, as this would involve knowledge of the inner workings of the host language compiler. This is a futile effort, as modern compilers span over 100'000 lines of code (the Glasgow Haskell Compiler has around 140'000 lines of code [**?**]).
- No host compiler or toolchain has to be installed, skipping another dependency.

2.1.2 Parsing the target language

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

A drawback is that many variants of BNF exist, such as EBNF [**?**], ABNF [**?**], ... Each of these variants have their own extra features, such as constructions for repetition or builtin basic forms. This is only superficial though: the underlying structure remains the same, simple search-and-replace can convert one dialect into another.

A new tool should thus use some form of BNF to declare the syntax, eventually with small differences which streamline the syntax even more, but with little extra features in order to simplify automatic reasoning[1].

No other lower-level details should be exposed to the language designer, parsing should be possible only using the BNF. The language designer should not have to deal with tokenization, the parser itself or the internal definition of the parsetree - this should all be builtin and hidden.

2.1.3 Executing the target language

A major goals is, of course, executing the target programming language. There are multiple ways to achieve this, which raises the question which one is best suited for prototyping languages. The concerns prototyping languages are:

- **Correctness** of the programming language. The metalanguage should help preventing bugs, thus have a clear and intuitive meaning.
- **Immediate and helpfull feedback**: when starting the program, error messages or output should be delivered as soon as possible.
- **Traceable**: it should be possible to see how a target program is executed, step by step. This helps the designer to gain insight in his language and what bugs are present.

Explecit *nongoals* for the tool are:

[1] Some of these extra features could be implemented as syntactic sugar.

- Running the target program fast. Special efforts optimizing the target language are out of scope, a small and simple implementation is favored.
- Compiling executables for any platform. Creating target binaries is not necessary for this research.[2]

This does point in the direction of building an interpreter for the programming language.

2.1.4 Typechecking the target language

If desired, a typechecker for the target language should be constructable as well. Preferably, no new metalanguage should be used to construct this algorithm. In other words, the typechecker should be build using the same linguistic metaconstructions as the interpreter.

Furthermore, **automatic correctness tests** should be added, which checks the typechecker in conjunction with interpreter.

2.1.5 Tooling

Practical aspects are important too - even the greatest tools lose users over unecessary 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 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 - 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, free, 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, 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. Most research tools also offer a **typesetting** module, which give a LaTeX-version of the language.

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), 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.

[2] This turned out to be possible. See section **??**.

```
1  grammar STFL;
2
3  bool     : 'True'
4           | 'False';
5
6  baseType         : 'Int'
7                   | 'Bool' ;
8
9  typeTerm         : baseType
10                  | '(' baseType ')';
11
12 type     : typeTerm '->' type
13          | typeTerm
14          ;
15
16 ID       : [a-z]+ ;
17
18 INT      : '0'..'9'+;
19
20 value    : bool
21          | INT;
22
23 e        : eL '+' e
24          | eL '::' type
25          | eL e
26          | eL ;
27
28 eL       : value
29          | ID
30          | '(' '\\' ID ':' type '.' e ')'
31          | 'If' e 'Then' e 'Else' e
32          | '(' e ')';
33
34 WS : [\t\r\n ]+ -> skip;
```

Figure 2.1: The grammar of STFL in ANTLR

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.

2.2.2 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 2.1

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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 2.2.

2.2.7 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 2.3.

2.2.8 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 2.4.

2.2.9 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 2.5 and 2.6. This language will be explained in more detail in chapter 3.

```
#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 2.2: Grammar definition and reduction rules for STFL in PLT-redex, compiled from the tutorial at [**?**]

```
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 2.3: 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 2.4: 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 2.5: 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 2.6: The reduction rule (with helper functions) of STFL in ALGT

2.3 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				✓	✓	✓	

2.4 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.

Chapter 3

ALGT in a Nutshell

In this part, we present **ALGT**, a tool and language to describe both syntax and semantics of aribitrary programming languages. This can be used to formally capture meaning of any language, formalizing them or, perhaps, gradualize them.

To introduce the ALGT-language, a small functional language (STFL) is formalized, giving a clear yet practical overview. This way, using ALGT to define a language should be clear.

This chapter does *not* give used command line arguments, an exhaustive overview of builtin functions, ... For this, we refer the reader to the tutorial and manual, which can be obtained by running `ALGT --manual-pdf`. This chapter neither gives algorithms used internally, such as the typechecker used on ALGT-languages, the proof searching algorithm, ... For these, we refer to the section about ALGT internals.

3.1 STFL

The *Simply Typed Functional Language* (STFL) is a small, functional language supporting integers, addition, booleans, if-then-else constructions and lambda abstractions, as visible in the examples (figure 3.1). Furthermore, variable declarations have a type, which can be `Bool`, `Int` or a function type. This typing is checked by a typechecker.

Due to its simplicity, it is widely used as example language in the field and thus well-understood throughout the community. This language will be gradualized in a later chapter.

Expression	End result
`1`	`1`
`True`	`True`
`If True Then 0 Else 1`	`0`
`41 + 1`	`42`
`(\x : Int . x + 1) 41`	`42`
`(\f : Int -> Int . f 41) (\x : Int -> Int . x + 1)`	`42`

Figure 3.1: Example expressions of STFL and their end result

3.2 Skeleton

A language is defined in a single `.language`-document, which is split in multiple sections: one for each aspect of the programming language. Each of these sections will be explored in depth. This results in a base skeleton of the language, as given below:

```
STFL # Name of the language
****

Syntax
======

# Syntax definitions

Functions
=========

# Rewriting rules, small helper functions

Relations
=========

# Declarations of which symbols are used

Rules
=====

# Natural deduction rules, defining operational semantics or typechecker

Properties
==========

# Automaticly checked properties
```

3.3 Syntax

The first step in formalizing a language is declaring *what the language looks like*, which is called the **syntax** of a language. Declaring the syntax can be done by writing BNF - a way to construct a context-free grammer. A context-free grammer can be used for two purposes: the first is creating all possible strings a language contains. On the other hand, the grammer can be used to deduce whether a given string is part of the language - and if it is, how this string is structured. This latter process is called *parsing*. ALGT can automatically construct a parser for the given BNF, which will turn the flat source code into a structured parsetree.

3.3.1 BNF

When formalizing a language syntax, the goal is to capture all possible strings that a program could be. This can be done with **production rules**. A production rule captures a *syntactic form* (a sublanguage) and consists of a name, followed by on or more options. An option is a sequence of parts, a part is a literal string or the name of another syntactic form:

```
nameOfRule      ::= otherForm
                | "literal"
                | otherForm "literal" otherForm1
                | ...
```

The syntactic form (or language) containing all boolean constants, can be captured with:

```
bool            ::= "True" | "False"
```

A language containing all integers has already been provided via a builtin. For practical reasons, it is given the name `int`:

```
int             ::= Number
```

The syntactic form containing all additions of two terms can now easily be captured:

```
addition        ::= int "+" int
```

Syntactic forms can be declared recursively as well, to declare more complex additions:

```
addition        ::= int "+" addition
```

Note that some syntactic forms are not allowed (such as empty syntactic forms or left recursive forms), this is more detailed in section 4.2.

3.3.2 STFL-syntax

In this format, the entire syntax for STFL can be formalized using multiple syntactic forms.

The first syntactic forms defined are types. Types are split into

- basetypes, containing only `"Bool"` and "Int"
- typeterms, containing basetypes and types between parentheses,
- the full types, containing either a function type with an `"->"` or a typeterm

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

The builtin constants `True` and `False` are defined as earlier introduced:

```
bool    ::= "True" | "False"
```

For integers and variables, the corresponding builtin values are used:

```
var     ::= Identifier
number  ::= Number
```

`number` and `bool` together form `value`, the expressions which are in their most simple form:

```
value   ::= bool | number
```

Another form term that can not be reduced any further is a bare lambda abstraction. A new syntacic form, `canon` is created, grouping all forms that can not be reduced any further:

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

Expressions are split into terms (`eL`) and full expressions (`e`):

```
e       ::= eL "+" e
        | eL "::" type
        | eL e
        | eL

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

A typing environment is provided as well. This is not part of the language itself, but will be used when typing programs:

```
typing                  ::= var ":" type
typingEnvironment       ::= typing "," typingEnvironment | "{}"
```

```
1  Resting string    stack
2  --------------    -----
3
4  "20 + 22"       ~ expr
5  "20 + 22"       ~ expr.0 (term "+" expr)
6  "20 + 22"       ~ term ; expr.0 ("+" expr)
7  "20 + 22"       ~ term.0 ("If" expr ...) ; expr.0 ("+" expr)
8  "20 + 22"       ~ term.1 ("(" "\" var ":" ...) ; expr.0 ("+" expr)
9  "20 + 22"       ~ term.2 (bool) ; expr.0 ("+" expr)
10 "20 + 22"       ~ bool ; term.2 (bool) ; expr.0 ("+" expr)
11 "20 + 22"       ~ bool.0 ("True") ; term.2 (bool) ; expr.0 ("+" expr)
12 "20 + 22"       ~ bool.1 ("False") ; term.2 (bool) ; expr.0 ("+" expr)
13 "20 + 22"       ~ term.3 (int) ; expr.0 ("+" expr)
14 "+ 22"          ~ expr.0 ("+" expr)
15 "22"            ~ expr.0 (expr)
16 "22"            ~ expr; expr.0 ()
17 ...
18 "22"            ~ term.3 (int) ; expr.2 (); expr.0 ()
```

Figure 3.2: The step-by-step parsing, based on a syntax definition; each line shows a step in the process. The first line gives the input string and syntactic form that it is parsed with. To parse `expr` , each of the choices is tried, the current choice being indicated with the index, the choice sequence written between parentheses. This sequence is parsed element per element; thus the first element of the sequence is moved on top of the stack. If the top element matches, the remainder string is shortened. If no element matches, the stack is popped and the next choice tried.

3.3.3 Parsing

Parsing is the process of structuring an input string, to construct a parsetree. This is the first step applied on a program in any compilation or interpretation process.

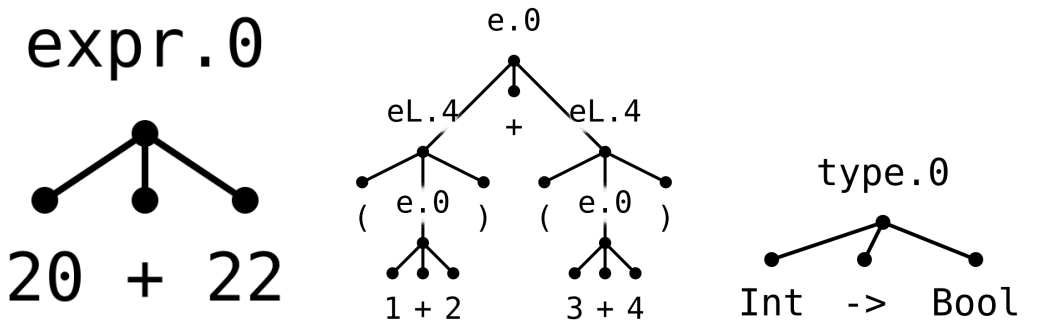
ALGT parses a target source code by trying to match a single syntactic form against the target program; eventually matching other subforms against parts of the input resulting in a parsetree.

When a string is parsed against syntactic form, the options in the definition of the syntactic form are tried, from left to right. The first option matching the string is used. An option, which is a sequence of either literals or other syntactic forms, is parsed by parsing element per element. A literal is parsed by comparing the head of the string to the literal itself, syntactic forms are parsed recursively.

In the case of `20 + 22` being parsed against `expr`, all the choices defining `expr` are tried, being `term "+" expr`, `term expr` and `term`. As parsing happens left to right, first `term "+" expr` is tried, implying the leftmost element of the sequence (`term`) is parsed with the string as input. After inspecting all the choices for term, the parser will use the last choice of term (`int`), which neatly matches `20`. The remainder string is now `+ 22`, which should be parsed against the rest of the sequence, `"+" expr`. The `"+"` in the sequence is now next to be tried, which matches the head of the string. The last `22` is parsed against `expr`. In order to parse `22`, the last choice of `expr`, thus `int` is used.

The resulting parsetree can be found in figure 3.3, together with some other examples.

This process is also denoted in figure 3.2.



(a) Parsetree of `20 + 22`

(b) Parsetree of `(1 + 2) + (3 + 4)`

(c) Parsetree of a function type, thus parsed with syntactic form `type`

Figure 3.3: Some example parsetrees

3.3.4 Conclusion

ALGT allows a concise yet accurate description of any language, through the use of BNF. This notation can then be interpreted in order to parse a target program file; resulting in the parsetree, a structure used to represent the target program.

3.4 Metafunctions

Existing parsetrees can be modified or rewritten by using **metafunctions**[1]. A metafunction receives one or more parsetrees as input and generates a new parsetree based on the input and function definition. These metafunctions are Turing-complete, so could be used to state the typechecker or interpreter for the target language. However, natural deduction (introduced in the next chapter) is a more structural way to achieve this. The metafunctions are however still an excellent tool to create smaller *helper functions*.

Metafunctions have the following form:

```
1 f : input1 -> input2 -> ... -> output
2 f(pattern1, pattern2, ...)      = someParseTree
3 f(pattern1', pattern2', ...)    = someParseTree'
```

The obligatory **signature** gives the name (`f`), followed by what syntactic forms the input parsetrees should have. The last element in the type[2] signature is the syntactic form of the parsetree that the function should return. ALGT effectivily *typechecks* functions, thus preventing the construction of parsetrees for which no syntactic form is defined.

The body of the functions consists of one or more **clauses**, containing one pattern for each input argument and an expression to construct the resulting parsetree.

Patterns have multiple purposes:

- First, they act as guard: if the input parsetree does not have the right form or contents, the match fails and the next clause in the function is activated. A pattern thus acts as an advanced `switch` of imperative languages.
- Second, they assign variables, which can be used to construct a new parsetree.
- Third, they can deconstruct large parsetrees, allowing finegrained pattern matching within the parsetrees' branches.
- Fourth, advanced searching and recombination behaviour is implemented in the form of evaluation contexts. This behaviour is explained and explored in section 3.5.5.

Expressions are the dual of patterns: where a pattern deconstructs, the expression does construct a parsetree; where the pattern assigns a variable, the expression will recall the variables value. Expressions are used on the right hand side of each clause, constructing the end result of the value.

An overview of all patterns and expressions can be found in the following tables:

Expr	As pattern
`x`	Captures the argument as the name. If multiple are used in the same pattern, the captured arguments should be the same or the match fails.
`_`	Captures the argument and ignores it
`42`	Argument should be exactly this number
`"Token"`	Argument should be exactly this string
`func(arg0, arg1, ...)`	Evaluates the function, matches if the argument equals the result. Can only use variables which are declared left of this pattern

[1] In this section, we will also use the term *function* to denote a *metafunction*. Under no condition, the term function refers to some entity of the target language.

[2] In this chapter, the term *type* is to be read as *the syntactic form a parsetree has*. It has nothing to do with the types defined in STFL. Types as defined within STFL will be denoted with `type`.

Expr	As pattern
`!func:type(arg0, ...)`	Evaluates the builtin function, matches if the argument equals the result. Can only use variables which are declared left of this pattern
`(expr or pattern:type)`	Check that the argument is an element of `type`
`e[expr or pattern]`	Matches the parsetree with `e`, searches a subtree in `e` which matches `pattern`
`a "b" (nested)`	Splits the parsetree in the appropriate parts, pattern matches the subparts

Expr	Name	As expression
`x`	Variable	Recalls the parsetree associated with this variable
`_`	Wildcard	*Not defined*
`42`	Number	This number
`"Token"`	Literal	This string
`func(arg0, arg1, ...)`	Function call	Evaluate this function
`!func:type(arg0, ...)`	Builtin function call	Evaluate this builtin function, let it return a `type`
`(expr or pattern:type)`	Ascription	Checks that an expression is of a type. Bit useless to use within expressions
`e[expr or pattern]`	Evaluation context	Replugs `expr` at the same place in `e`. Only works if `e` was created with an evaluation context
`a "b" (nested)`	Sequence	Builds the parsetree

3.4.1 Domain and codomain

With these expressions and patterns, it is possible to make metafunctions extracting the domain and codomain of a function `type` (in STFL). These will be used in the typechecker in the following chapter. `domain` and `codomain` are defined in the following way:

```
1  dom                          : type -> typeTerm
2  dom("(" T1 ")")              = dom(T1)
3  dom(("(" T1 ")") "->" T2)    = T1
4  dom(T1 "->" T2)              = T1
5
6
7  cod                          : type -> type
8  cod("(" T2 ")")              = cod(T2)
9  cod(T1 "->" ("(" T2 ")"))    = T2
10 cod(T1 "->" T2)              = T2
```

Recall the parsetree generated by parsing `Int -> Bool` against `type`. If this parsetree were used as input into the `domain` function, it would fail to match the first pattern (as the parsetree does not contain parentheses) nor would it match the second pattern (again are parentheses needed). The third pattern matches, by assigning `T1` and `T2`, as can seen in figure 3.4. `T1` is extracted and returned by `domain`, which is, by definition the domain of the `type`.

Analogously, the more advanced parsetree representing `Int -> Bool -> Bool` will be deconstructed the same way, as visible in figure 3.5, again capturing the domain within `T1`.

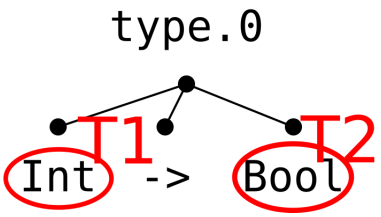


Figure 3.4: The pattern match of `Int -> Bool` against pattern `T1 "->" T2`, used in the `domain` function. The domain will be captured within `T1`

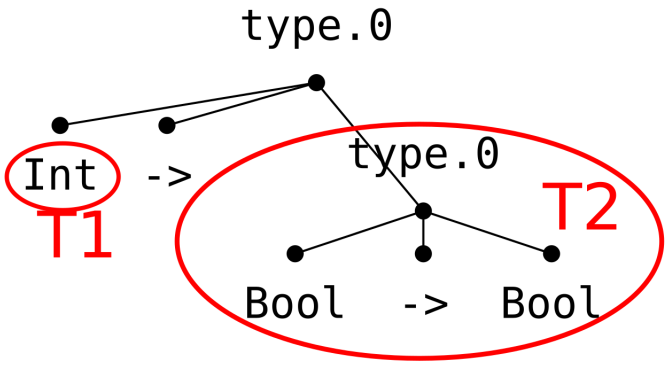


Figure 3.5: The pattern match of `Int -> Bool -> Bool` against pattern `T1 "->" T2`, used in the `domain` function. The domain will again be captured within `T1`

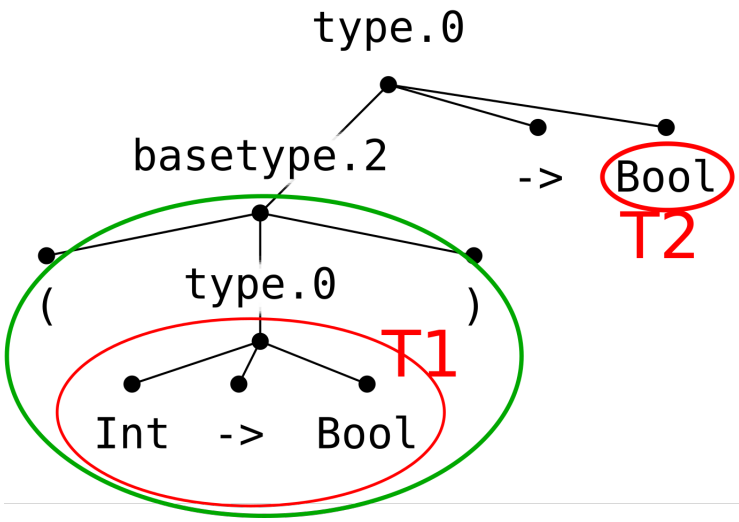


Figure 3.6: The variable assignment after pattern matching `(Int -> Bool) -> Bool` against pattern `("(" T1 ")") "->" T2`. The entire subpattern `"(" T1 ")"` is circled in green, where `T1` will again capture the domain.

The deconstructing behaviour of patterns can be observed when `(Int -> Bool) -> Bool` is used as argument for `domain`. It matches the second clause, deconstructing the part left of the arrow, namely `(Int -> Bool)`, and matching it against the embedded pattern `"(" T1 ")"`, as visualised in figure 3.6, capturing the domain *without parentheses* in `T1`.

3.4.2 Equate

Another usefull function is `equate`, which compares two values for equality and is undefined if the values are not the same. Its implementation is the following:

```
1 equate  : type -> type -> type
2 equate(T, T)    = T
```

This function uses two patterns which bind the same variable, which is perfectly allowed. When the function is applied and the variable store is constructed, the pattern matcher will detect if a variable is bound to multiple values. If such a misbinding is detected, the pattern fails and falls through. If all arguments bound to the variable `T` are the same, the match succeeds.

3.4.3 Safety checks

When the language is loaded by ALGT, the metafunctions are checked for possible errors:

- Patterns are typechecked, detecting invalid deconstructions
- Expressions are typechecked, preventing the construction of malformed parsetrees, thus parsetrees for which no syntactical definition exists
- The patterns are aggregated and ALGT checks that no pattern can fallthrough
- The aggregated patterns are checked that at least one argument combination can match the pattern, so that no clause is dead.

These checks catch many small inconsistencies the language designer might make, catching many bugs.

The checks catching liveness and fallthrough are explained in the chapter about abstract interpretation, in particular section 4.5.4.

3.4.4 Conclusion

Metafunctions give a concise, typesafe way to transform parsetrees. The many checks, such as wrong types, liveness and fallthrough perform the first sanity checks and catch many bugs beforehand.

3.5 Natural deduction

In this section, the syntactic forms of STFL are given meaning by rewriting a parsetree to its canonical form. After a short exposure on different ways to inject semantics to a program, operational semantics are implemented for STFL.

3.5.1 Giving meaning to a program

In section 1.1.2, a definition of semantics and multiple approaches to describing them were given.

Recall that the meaning of a program is the effect that it has on the world. Four frameworks to specify semantics were described:

- Translation to another language
- Denotational semantics
- Axiomatic semantics
- Structural operational semantics

While all of the above semantical approaches are possible within ALGT, structural operational semantics (or operational semantics for short) is the easiest and most practical approach for this dissertation. The main ingredient, the syntax of the language, is already present; only the transformations of the parsetree should be denoted. As STFL is a functional language, there is no need for a state to be modeled.

3.5.2 Semantics as relations

All these semantics, especially structural operational semantics, are relations:

- Translation and denotational semantics revolves around the relation between two languages
- Structural operational semantics for functional languages revolves around the relation between two expressions - the original expression and the expression after evaluation
- Operational semantics for an imperative program revolves around the relation around the state before, a statement and a state after.
- Axiomatic semantics relate the properties before and after execution of an imperatie

All these relations can be constructed using natural deduction rules in a straightforward and structured way. The most

In the rest of this chapter, natural deduction is used to construct a structur semantic for STFL, followed by a typechecker.

3.5.3 Declaring relations

Smallstep is the relation between STFL-expressions, which ties an expression to another expression which is smaller, but has the same value. As example, smallstep rewrites expressions as `1 + 1` into `2` or `If True Then 41 + 1 Else 0` into `41 + 1`. This relation will be denoted with the symbol $\rightarrow$. As this relation is between two expressions, it lies within `expr` $\times$`expr`.

Defining this relation within ALGT is done in two steps. First, the relation is declared in the `Relations`-section, afterwards the implementation is given in the `Rules` section.

The declaration of smallstep in ALGT is as following:

```
1 Relations
2 ===========
3
4 (→)     : e (in), e (out)       Pronounced as "smallstep"
```

Lines 1 and 2 give the `Relation`-header, indicitating that the following lines will contain relation declarations. The actual declaration is in line 4.

First, the symbol for the relation is given, between parentheses: `(→)`. Then, the types of the arguments are given, by `: expr (in), expr (out)`, denoting that $\rightarrow$ is a relation in `expr` $\times$`expr`. Each argument has a mode, one of `in` or `out`, written between parentheses. This is to help the tool when proving the relation: given `1 + 1` as first argument, the relation can easily deduce that this is rewritten to `2`. However, given `2` as second argument, it is hard to deduce that this was the result of rewriting `1 + 1`, as there are infinitely many expressions yielding `2`.

Relations might have one, two or more arguments, of which at least one should be an input argument[3]. Relations with no output arguments are allowed, an example of this would be a predicate checking for equality.

The last part, `Pronounced as "smallstep"` is documentation. It serves as human readable name, hinting the role of the relation within the language for users of the programming language which are not familiar with commonly used symbols. While this is optional, it is strongly recommended to write: it gives a new user a prononciation for the relation and, even more important, a term which can be searched for in a search engine. Symbols are notoriously hard to search.

3.5.4 Natural deduction rules

The declaration of a relation does not state anything about the actual implementation of this relation. This implementation is given by natural deduction rules. Each natural deduction rule focuses on a single aspect of the relation. In this sections, how to construct natural deduction rules is explained in detail.

Simple natural deduction rules

A simple natural deduction rule is given to ALGT in the following form:

```
1 ---------------------------------- [Rule0Name]
2  (relation) arg0, arg1, arg2, ...
```

The relation can also be written in an infix way:

```
1 ---------------------------------- [Rule0Name]
2  arg0 relation arg1, arg2, ...
```

The most important part of a rule is written below the line, which states that `(arg0, arg1, arg2, ...)` is in `relation`. The arguments can be advanced patterns/expressions, just as seen with functions. The expression on an input argument location is treated as a pattern, which will match the input parsetree and construct a variable store. Expressions on output argument locations use this variable store to construct the output. This is illustrated in figure 3.8, where a rule introduced in figure 3.7 is used.

[3]There is no technical restriction forcing a relation to have at least one input argument. However, a relation with only output arguments will always have exactly the same output. Such a thing is called *a constant* and can be written *without* a relation. This renders a relation with only input arguments quite useless.

```
 Relations
===========

(~>) : expr (in), expr (out)    Pronounced as "example relation"


 Rules
=======

------------------ [Example Rule]
 a "+" b ~> !plus(a, b)
```

Figure 3.7: Example relation containing a single rule, which calls a metafunction

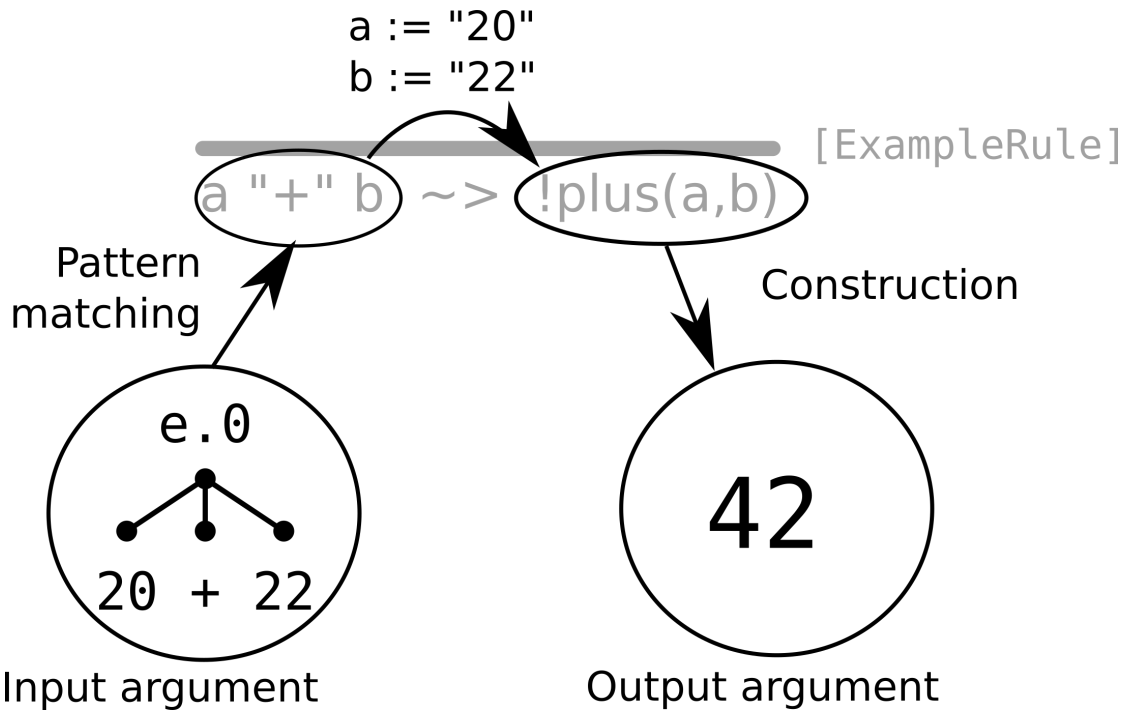


Figure 3.8: Application of the example rule, introduced in figure 3.7. The flow of information through this simple rule is denoted: the input argument is patterned matched, resulting in a variable store. Using this variable store, the output argument is constructed.

Natural deduction rules with predicates

In some cases, extra conditions apply before a tuple is part of a relation. Extra conditions could be that:

- Two parsetrees should be the same, or two parts in a parsetree should be the same
- A parsetree should be of a certain syntacic form
- Some relation between the arguments should hold.

This can be forced by using predicates, which are written above the line seperated by tab characters:

```
 (rel) arg1 arg2        arg0:form       arg0 = arg2
------------------------------------------------------ [Rule1Name]
 (relation) arg0, arg1, arg2, ...
```

This rules is pronounced as _ if `(rel) arg1, arg2` holds, if `arg0` is a `form` and `arg0 = arg2`, then `(relation) arg0, arg1, arg2` holds.

Predicates are evaluated from left to right, where a relation predicate might introduce new variables in the variable store. This information flow is illustrated in 3.10.

```
 Relations
===========

(~>) : expr (in), expr (out)    Pronounced as "example relation"

 Rules
=======

------------------------ [Example Rule]
 a "+" b ~> !plus(a, b)

 b:bool         b = "True"      e0 ~> e1
---------------------------------------- [Example Predicate Rule]
 "If" b "Then" e0 "Else" e ~> e1
```

Figure 3.9: Example relation containing a predicate bounded rule, which is applied in 3.10

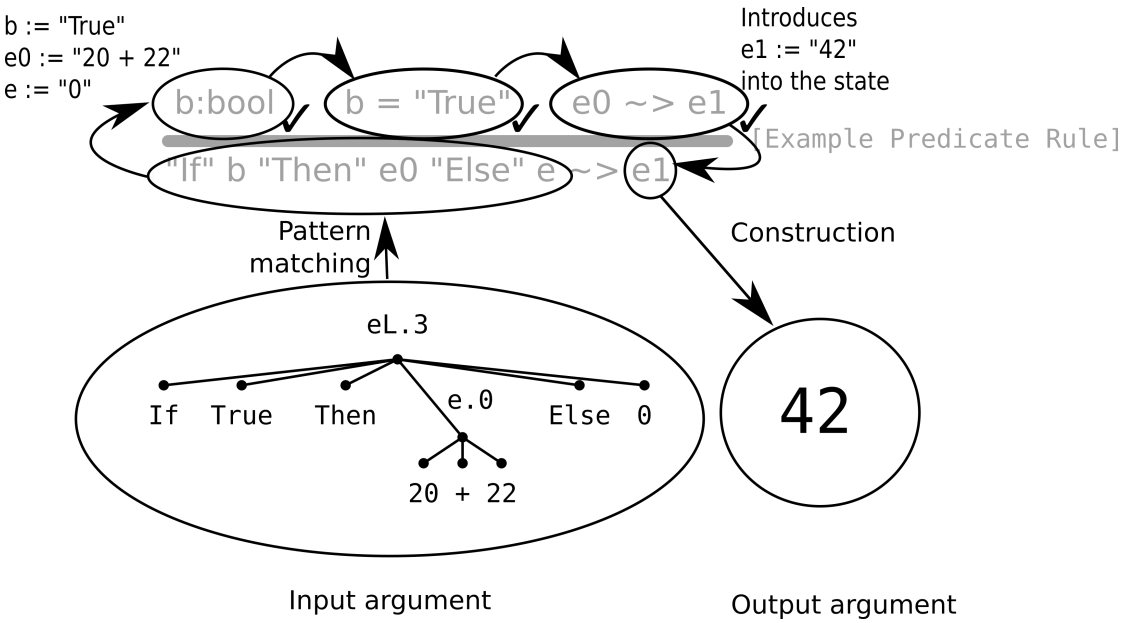


Figure 3.10: Application of a rule with an input argument, an output argument and predicates. The pattern matching of the input argument generates initial the state in the left upper corner, namely `b:="True", e0 := "20 + 22", e := "0"` . Each of the predicates is tested with this state in a left to right fashion. The last predicate, `e0 > e1` applies rule *ExampleRule* with `e0` as input argument and `e1` as unbound variable, as seen in 3.8. This results in `e1` to be introduced in the state and `e1` to be bound to the resulting value, namely `"42"` . This is used to construct the output argument.

3.5.5 Defining smallstep

With these tools introduced in the previous chapter, it is feasible to define *smallstep*. Each facet of this relation will be described by a single natural deduction rule.

Remember that smallstep rewrites a small piece of a parsetree in the smallest possible amount. In other words, for each syntactic form, a the simplification will be given. This does not guarantee that the parsetree is fully evaluated; rather, smallstep only makes the tree *smaller* while preserving its value.

Deduction rules for the following syntacic forms are given:

- If-expressions
- Parentheses
- Addition
- Type ascription
- Lambda abstraction and application

Furthermore, a convergence rule for nested expressions is added.

If-then-else and parentheses

For starters, the rule evaluating `If True Then ... Else ...` can be easily implemented:

```
1
2 ---------------------------------------          [EvalIfTrue]
3  "If" "True" "Then" e1 "Else" e2 → e1
```

This rule states that `("If" "True" "Then" e1 "Else" e2, e1)` is an element of the relation →. In other words, *`"If" "True" "Then" e1 "Else" e2` is rewritten to `e1`.*

Analogously, the case for `False` is implemented:

```
1
2 ----------------------------------------         [EvalIfFalse]
3  "If" "False" "Then" e1 "Else" e2 → e2
```

Another straightforward rule is the removal of parentheses:

```
1
2 ----------------                              [EvalParens]
3  "(" e ")" → e
```

Addition

The rule `EvalPlus` reduces the syntactic form `n1 "+" n2` into the actual sum of the numbers, using the builtin function `!plus`. However, this builtin function can only handle `Number`s; a parsetree containing a richer expression can't be handled by `!plus`. This is why two additional predicates are added, checking that `n1` and `n2` are of syntactic form `Number`.

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

Type ascription

Type ascription is the syntactic form checking that an expression is of a certain type. If this is the case, evaluation continues with the nested evaluation. If not, the execution of the program halts.

As predicate, the typechecker defined in the next part is used, which is denoted by the relation `(::)`. This relation infers, for a given `e`, the corresponding type `T`. The rule is defined as following:

```
1 e  ::  T
2 ------------------------                [EvalAscr]
3 e "::" T → e
```

In order to gradualize this language later on, a self-defined equality relation is used. This equality `==` will be replaced with *is consistent with* `~` when gradualizing, resulting in:

```
1 e :: T0        T == T0
2 ------------------------                [EvalAscr]
3 e "::" T → e
```

Applying lambdas

The last syntactic form to handle are applied lambda abstractions, such as `(x : Int . x + 1) 41`. The crux of this transformation lies in the substitution of the variable `x` by the argument everywhere in the body. Substitution can be done with the builtin function `!subs`.

The argument should have the expected type, for which the predicate `arg :: T` is added. The argument should also be fully evaluated in order to have strict semantics. This is checked by the predicate `arg:value`, giving the rule:

```
1 arg:canon      arg :: TArg     TArg == TExp
2 ------------------------------------------------ [EvalLamApp]
3 ("(" "\\" var ":" TExp "." e ")") arg → !subs:e(var, arg, e)
```

Convergence

There is still a problem with the relation for now: the given evaluation rules can't handle nested expressions, such as `1 + (2 + 39)`. No rule exists yet which would evaluate this expression to `1 + 41`.

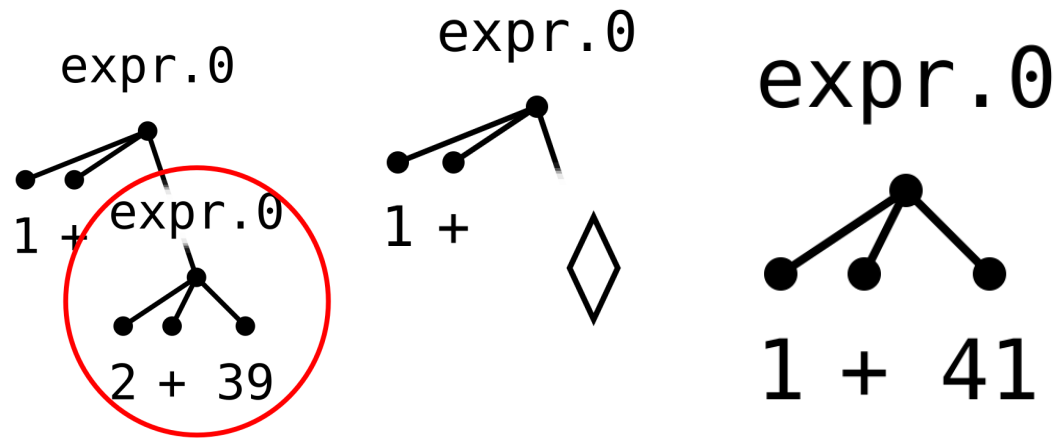
It is rather cumbersome to introduce rules for each position where a syntactic form might be reduced. For `+`, this would need two extra rules (`e0 "+" e →e1 "+" e` and `e "+" e0 →e "+" e1`), `If`-expressions would need an additional three rules, ... This clearly does not scale.

A scaling solution is the following convergence rule:

```
1 e0 → e1
2 -----------------                        [EvalCtx]
3 e[e0] → e[e1]
```

This rule uses an *evaluation context*: `e[e0]`. This pattern will capture the entire input as `e` and search a subtree matching the nested expression `e0` *fullfilling all the predicates*. In this case, an `e0` is searched so that `e0` can be evaluated to `e1`.

When the evaluation context `e[e1]` is used to construct a new parsetree (the output argument), the original parsetree `e` is modified. Where the subtree `e0` was found, the new value `e1` is plugged back, as visible in figure 3.11.



(a) Parsetree of `1 + (2 + 39)`, which is pattern matched against `e[e0]`; the subtree `2 + 39` is matched against `e0`

(b) The parsetree `e` with `e0` replaced by *a hole*. This hole will later be filled

(c) The parsetree `e[e1]`, thus the parsetree with the hole filled with the evaluated subtree

Figure 3.11: An evaluation context where a subtree is replaced by its corresponding evluated expression

Overview

The semantics of the STFL language can be captured in 7 straightforward natural deduction rules, in a straightforward and human readable format. For reference, these rules are:

```
 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

```

```
29
30 ---------------------------------------        [EvalIfFalse]
31  "If" "False" "Then" e1 "Else" e2 → e2
32
33
34  arg:canon      arg :: TArg     TArg == TExp
35 ---------------------------------------------- [EvalLamApp]
36  ("(" "\\" var ":" TExp "." e ")") arg → !subs:e(var, arg, e)
```

The relation → needs a single input argument, so it might be run against an example expression, such as `1 + 2 + 3`.

```
# 1 + 2 + 3 applied to →
# Proof weight: 4, proof depth: 3


2 : Number    3 : Number
----------------------- [EvalPlus]
2 + 3 → 5
------------------------ [EvalCtx]
1 + 2 + 3 → 1 + 5
```

Running → over a lambda abstraction gives the following result:

```
# (\x : Int . x + 1) 41 applied to →
# Proof weight: 7, proof depth: 4


              41 : number
              ------------ [Tnumber]
              {} ⊢ 41, Int                          equate(T1, T2) = Int = T1
              ---------------------- [TEmptyCtx]    -------------------- [Eq]
41 : canon    41 :: Int                             Int == Int
-------------------------------------------------------------------- [EvalLamApp]
( \ x : Int . x + 1 ) 41 → 41 + 1
```

3.5.6 Typechecker

A typechecker checks expressions for constructions which don't make sense, such as `1 + True`. This is tremendously usefull to catch errors in a static way, before even running the program.

In this section, a typechecker for STFL is constructed, using the same notation as introduced in the previous section. Just like the relation $\rightarrow$, which related two expressions, a relation `::` is defined which relates expression to its type. The relation `::` is thus a relation in `expr` $\times$`type`.

Examples of elements in this relation are:

expression	type
`True`	`Bool`
`If True Then 0 Else 1`	`Int`
`(\ x : Int . x + 1)`	`Int -> Int`
`1 + 1`	`Int`
`1 + True`	*undefined, type error*

However, this relation can't handle variables. When a variable is declared, its type should be saved somehow and passed as argument with the relation. Saving the types of declared variables is done by keeping a **typing environment**, which is a syntacical form acting as list. As a reminder, it is defined as:

```
1 typing                ::= var ":" type
2 typingEnvironment     ::= typing "," typingEnvironment | "{}"
```

A new relation `(⊢)` is introduced, taking the expression to type and a typing environment to deduce the type of an expression. Some example elements in this relation are:

expression	Typing environment	type
`True`	`{}`	`Bool`
`x`	`x : Bool, {}`	`Bool`
`If True Then 0 Else 1`	`x : Bool, {}`	`Int`
`(\ x : Int . x + 1)`	`{}`	`Int -> Int`
`1 + 1`	`y : Int, {}`	`Int`
`1 + True`	`{}`	*undefined, type error*

3.5.7 Declaration of :: and $\vdash$

Implementing these relations begins (again) with declaring the relations.

Both `::` and $\vdash$ have an `expr` and a `type` argument. In both cases, `expr` can't be of mode `out`, as an infinite number of expressions exist for any given type. On the other hand, each given expression can only have one corresponding type, so the `type`-argument can be of mode `out`. The typing environment argument has mode `in` just as well; as infinite many typing environments might lead to a correct typing, namely all environments containing unrelated variables.

The actual declaration thus is as following:

```
1 (⊢)     : typingEnvironment (in), e (in), type (out)    Pronounced as "context entails typing"
2 (::)    : e (in), type (out)    Pronounced as "type in empty context"
```

3.5.8 Definition of ⊢

The heavy lifting is done by ⊢, which will try to deduce a type for each syntacic concstruction, resulting in a single natural deduction rule for each of them. As a reminder, the following syntacic forms exist in STFL and are typed:

- Booleans
- Integers
- Expressions within parens
- Addition
- If-expressions
- Type ascription
- Variables
- Lambda abstractions
- Function application

Each of these will get a typing rule in the following paragraphs. In these rules, Γ will always denote the typing environment. Checking types for equality is done with a custom equality relation `==`, which simply passes its arguments into the function `equate`, which was defined earlier:

```
1 equate(T1, T2)
2 ---------------- [Eq]
3 T1 == T2
```

Typing booleans

Basic booleans, such as `True` and `False` can be typed right away:

```
1 b:bool
2 ----------------        [Tbool]
3 Γ ⊢ b, "Bool"
```

Here, Γ denotes the typing environment (which is not used in this rule); `b` is the expression and `"Bool"` is the type of that expression. In order to force that `b` is only `"True"` or `"False"`, the predicate `b:bool` is used.

Typing integers

The rule typing integers is completely analogously, with a predicate checking that `n` is a `number`:

```
1 n:number
2 ---------------         [Tnumber]
3 Γ ⊢ n, "Int"
```

Typing parens

An expression of the form `(e)` has the same type as tye enclosed expression `e`. This can be expressed with a predicate typing the enclosed expression:

```
1 Γ ⊢ e, T
2 -------------------     [TParens]
3 Γ ⊢ "(" e ")", T
```

Typing addition

Just like a number, an addition is always an `Int`. There is a catch though, namely that both arguments should be `Int` too. This is stated in the predicates of the rule:

```
1 Γ ⊢ n1, Int1   Γ ⊢ n2, Int2    Int1 == "Int"   Int2 == "Int"
2 ------------------------------------------------------------ [TPlus]
3 Γ ⊢ n1 "+" n2, "Int"
```

Typing if

A functional `If`-expression has the same type as the expressions it might return. This introduces a constraint: namely that the expression in the if-branch has the same type as the expression in the else-branch. Furthermore, the condition should be a boolean, resulting in the four predicates of the rule:

```
1 Γ ⊢ c, TCond   Γ ⊢ e1, T0      Γ ⊢ e2, T1      TCond == "Bool"          T0 == T1
2 ------------------------------------------------------------------------------  [TIf]
3 Γ ⊢ "If" c "Then" e1 "Else" e2, T0
```

Type ascription

Typing a type ascription boils down to typing the nested expression and checking that the nested expression has the same type as is asserted:

```
1 Γ ⊢ e, T'      T' == T
2 ---------------------- [TAscr]
3 Γ ⊢ e "::" T, T'
```

Variables

Variables are typed by searching the corresponding typing in the typing environment. This searching is implemented by the evaluation context, as `Γ[x ":" T]` will search a subtree matching a variable named `x` in the store. When found, the type of `x` will be bound in `T`:

```
1 --------------------    [Tx]
2 Γ[ x ":" T ] ⊢ x, T
```

These variable typings are introduced in the environment by lambda abstractions.

Lambda abstractions

The type of a lambda abstraction is a function type from the type of the argument to the type of the body.

The type of the argument is explicitly given and can be immediatly used. The type of the body should be calculated, which is done in the predicate. Note that the typing of the body considers the newly introduced variable, by appending it into the typing environment, before passing it to the relation:

```
1 ((x ":" T1) "," Γ) ⊢ e, T2
2 ------------------------------------------------------ [TLambda]
3 Γ ⊢ "(" "\\" x ":" T1 "." e ")", T1 "->" T2
```

Function application

The last syntacic form to type is function application. In order to type an application, both the function and argument are typed in the predicates. To obtain the type of an application, the codomain of the function type is used. Luckily, a helper function was introduced earlier which calculates exactly this. At last, the argument should be of the expected type, namely the domain of the function, resulting in an extra predicate:

```
1 Γ ⊢ e1, Tfunc  Γ ⊢ e2, Targ    Targ == dom(Tfunc)
2 ------------------------------------------------     [Tapp]
3 Γ ⊢ e1 e2, cod(Tfunc)
```

3.5.9 Definition of ::

As :: is essentialy the same as ⊢ with an empty type environment, :: is defined in terms of ⊢. The expression argument is passed to ⊢, together with the empty type environment {}:

```
1 "{}" ⊢ e, T
2 -----------     [TEmptyCtx]
3 e :: T
```

Typing expressions

With all the rules defined, expressions can be typed. This results in a derivation[4] of why an expression has a certain type:

```
# 1 + 2 + 3 applied to ::
# Proof weight: 17, proof depth: 5


                  2 : number         3 : number
                  --- [TNumber]     --- [TNumber]
 1 : number       {} ⊢ 2, Int       {} ⊢ 3, Int
--- [TNumber]     -------------------- [TAddition]
 {} ⊢ 1, Int      {} ⊢ 2 + 3, Int
------------------------------------- [TPlus]
{} ⊢ 1 + 2 + 3, Int
---------------------------------- [TEmptyCtx]
1 + 2 + 3 :: Int
```

```
# (\ x: Int. x + 1) 41 applied to ::
# Proof weight: 15, proof depth: 6


                          1 : number
----------------- [Tx]    ------------ [Tnumber]
x : Int , {} ⊢ x, Int     x : Int , {} ⊢ 1, Int
---------------------------------------- [TPlus]
x : Int , {} ⊢ x + 1, Int                          41 : number
---------------------------- [TLambda]             --- [Tnumber]
{} ⊢ ( \ x : Int . x + 1 ), Int -> Int             {} ⊢ 41, Int
------------------------------------------------------------ [Tapp]
{} ⊢ ( \ x : Int . x + 1 ) 41, Int
---------------------------------------------------- [TEmptyCtx]
( \ x : Int . x + 1 ) 41 :: Int
```

[4] For printing, the derivations of == relations are omitted.

3.5.10 Overview

These ten natural deduction rules describe the entire typechecker. For reference, they are all stated here together with the definition of the convenience relation `::` :

```
 "{}" ⊢ e, T
-----------     [TEmptyCtx]
 e :: T



 n:number
---------------          [Tnumber]
 Γ ⊢ n, "Int"


 b:bool
----------------         [Tbool]
 Γ ⊢ b, "Bool"



 Γ ⊢ e, T
-------------------      [TParens]
 Γ ⊢ "(" e ")", T


 Γ ⊢ e, T'      T' == T
----------------------- [TAscr]
 Γ ⊢ e "::" T, T'



--------------------     [Tx]
 Γ[ x ":" T ] ⊢ x, T


 Γ ⊢ n1, Int1   Γ ⊢ n2, Int2    Int1 == "Int"   Int2 == "Int"
------------------------------------------------------------- [TPlus]
 Γ ⊢ n1 "+" n2, "Int"


 Γ ⊢ c, TCond   Γ ⊢ e1, T0      Γ ⊢ e2, T1      TCond == "Bool"          T0 == T1
-------------------------------------------------------------------------  [TIf]
 Γ ⊢ "If" c "Then" e1 "Else" e2, T0



 ((x ":" T1) "," Γ) ⊢ e, T2
------------------------------------------------------ [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", T1 "->" T2



 Γ ⊢ e1, Tfunc  Γ ⊢ e2, Targ    Targ == dom(Tfunc)
-------------------------------------------------     [Tapp]
 Γ ⊢ e1 e2, cod(Tfunc)
```

3.5.11 Conclusion

Naturad deduction rules offer a comprehensible and practical way to capture operational semantics. It is no surprise they are widely adopted by the language design community. Automating the evaluation of such rules offers huge benefits, such as catching type errors and immediately getting an implementation from the specification, removing all ambiguities.

3.6 Declaring and checking properties

The third important aspect of a language, apart from syntax and semantics, are the properties it obeys. These properties offer strong guarantees about the language and semantics. In the next paragraphs, two important properties of STFL are stated and added to the language definition. ALGT will quickcheck them.

3.6.1 Progress and Preservation

The first important guarantee is that a *well typed expression* can always be evaluated. Without this property, called **Progress**, using a typechecker would be useless; as some well-typed expression would still crash or hang. Another important property, called **Preservation**, is that an expression of type `T` is still of type `T` after evaluation. While this property is not surprising, it is important to formally state this. Together, these properties imply that a well-typed expression can be evaluated to a new, well-typed expression of the same type.

As this is an important aspect as well, ALGT offers a way to incorporate these properties into the language definition. These properties are automatically checked by randomly generated examples or might be tested against an example program. Sadly, it is not possible to automatically proof arbitrary properties, so some cases for which the property does not hold might slip through.

3.6.2 Stating Preservation

Preservation states that, *if an expression `e0` has type `T` and if `e0` can be evaluated to `e1`, then `e1` should be of type `T` too.* Note that all these conditions can be captured with the earlier definied relations: *if `e0 :: T` and `e0 →e1`, then `e1 :: T`.* This semantics are pretty close to a natural deduction rule with predicates, but instead of adding the consequent to the relation, checking whether the conclusion holds does suffice.

The natural deduction rule syntax is thus reused to state properties:

```
1 | e0 :: T          e0 → e1
2 | -------------------------- [Preservation]
3 |          e1 :: T
```

3.6.3 Stating Progress

Progress states that, *if an expression `e` can be typed (as type `T`), then `e` is either fully evaluated or `e` can be evaluated further.* This to can be stated in terms of the earlier defined relations: *if `e :: T`, then `e:value` or `e →e1`.*

In order to denote the choice, a new syntax is introduced, using a `|` (a vertical bar). This syntax is not allowed in natural deduction rules defining relations. Furthermore, allowing predicates of the form `e:value` is another difference with the syntax used to define rules.

The property can be stated as:

```
1          e0 :: T
2 -------------------------- [Progress]
3  e0:value     |      e0 → e1
```

3.6.4 Testing properties

ALGT will test these properties automatically each time the language definition is loaded. In order to test a property, a testing sample of is generated, by calculating what inputs are needed for the property (the input for Preservation is `e0`, Progress needs `e` to be generated). For those inputs, parsetrees are randomly generated and the property is tested.

These properties offer an extra guarantee about the language created, again in a non-ambiguous and machine-checkable way, a feature increasing the language designers productivity.

It is possible to print those proofs, as given below:

```
# Property Preservation statisfied with assignment
# {T --> Int, e0 --> 1 + 2, e1 --> 3}
  # Predicate satisfied:
  # e0 :: T

    1 : number                2 : number
    ----------- [Tnumber]     ----------- [Tnumber]
    {} ⊢ 1, Int               {} ⊢ 2, Int
    ------------------------------------ [TPlus]
    {} ⊢ 1 + 2, Int
    ---------------------------------- [TEmptyCtx]
    1 + 2 :: Int

  # Predicate satisfied:
  # e0 → e1

    1 : Number    2 : Number
    ------------------------ [EvalPlus]
    1 + 2 → 3



  # Satisfies a possible conclusion:
  # e1 :: T

  3 : number
  ----------- [Tnumber]
  {} ⊢ 3, Int
  -------------------- [TEmptyCtx]
  3 :: Int


Property Progress holds for given examples
Property successfull
# Property Progress statisfied with assignment {T --> Int, e0 --> 1 + 2}
  # Predicate satisfied:
  # e0 :: T

    1 : number                2 : number
    ----------- [Tnumber]     ----------- [Tnumber]
    {} ⊢ 1, Int               {} ⊢ 2, Int
    ------------------------------------ [TPlus]
    {} ⊢ 1 + 2, Int
    ---------------------------------- [TEmptyCtx]
    1 + 2 :: Int
```

```
# Satisfies a possible conclusion:
# e0 → e1

1 : Number    2 : Number
------------------------ [EvalPlus]
1 + 2 → 3
```

[style=terminal]

3.7 Conclusion

ALGT allows a versatile, human readable and portable format to define programming languages, supporting a wide range of operations, such as parsing, running and typechecking target languages. On top, properties about the language in general can be stated and tested.

Each funcamental aspect of the language is given his place, giving rise to a formally correct and machine-checkable language definition. No ambiguities can possibly exist, creating a highly exchangable format. By keeping the focus on readability, no host-language boiler plate should be learned on written, creating a language agnostic tool to exchange new programming languages.

Chapter 4

Syntax Driven Abstract Interpretation

In this section, functions on parsetrees are converted into functions over sets of parsetrees. This is useful to algorithmically analyze these functions, which will help to gradualize them. The technique to convert these metafunctions, called **abstract interpretation** is dissected in this part, which is organized as following:

- First, we'll work out **what abstract interpretation is**, with simple examples followed by its desired properties.
- Then, we work out what **properties a syntax** has.
- With these, we develop an **efficient representation** to capture infinite sets of parsetrees.
- Afterwards, **operations on these setrepresentations** are stated.
- As last, the metafunctions are actually lifted to **metafunctions over sets**

4.1 Abstract interpretation

Per Rice's theorem, it is generally impossible to make precise statements about all programs. However, making useful statements about some programs is feasable. Cousot introduces a suitable framework, named **abstract interpretation**: "A program denotes computations in some domain of objects. Abstract interpretation of programs consists in using that denotation to describe computations in *another domain of abstract objects*, so that the results of the abstract computations give some information on the actual computation" [**?**].

This principle can best be illustrated, for which the he successor function (as defined in figure 4.1) is a prime example. The function normally operates in the domain of *integer numbers*, as `succ` applied on `1` yields `2`; `succ -1` yields `0`.

But `succ` might also be applied on *signs*: the symbols `+`, `-` or `0` are used instead of integers, where `+` represents all strictly positive numbers and `-` represents all strictly negative numbers. These symbols are used to perform the computation, giving rise to a computation in the abstract domain of signs.

```
1 succ n  = n + 1
```

Figure 4.1: Definition of the *successor* function, defined for natural numbers

4.1.1 The rule of signs

A positive number which is increased is always positive. Thus, per rule of signs `+` $+1$ is equal to `+`. The calculation of `succ +` thus yields `+`.

On the other hand, a negative number which is increased by one, might be negative, but might be zero too. `-` $+1$ thus results in both `0` and `-`. The result is that applying `succ` to a negative number gives less precise information.

The question now arises how to deal with less precise information. One option might be to fail and indicate that no information could be deduced at all. Another option is to introduce exra symbols representing the unions of `+`, `-` and `0`. The symbol representing the negative numbers (including zero) would be `0-`, analogously does `0+` represent the positive numbers (including zero). Both the strictly negative and strictly positive numbers are represented by `+-`. At last, the union of all numbers is represented with $\top$.

These symbols represent a set, where the set represented by `+` (the strictly positive numbers) are embedded in the set represented by `0+` (the positive numbers). This *embedding* relation forms a lattice, as each two symbols have a symbol embedding the union of both, as can be seen in 4.2.

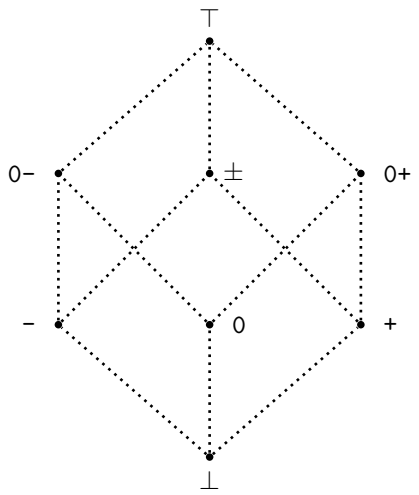


Figure 4.2: The symbols representing sets. Some sets, such as `+` are embedded in others, such as `0+`. These embeddings form a lattice.

Concretization and abstraction

The meaning of `succ -` *giving* `0-` is intuitionally clear: *the successor of a negative number is either negative or zero.* More formally, it can be stated that, *given a negative number,* `succ` *will give an element from* $\{n|n \leq 0\}$. The meaning of `0-` is formalized by the **concretization** function γ, which translates from the abstract domain to the concrete domain:

$$\begin{aligned}
\gamma(\texttt{ - }) &= \{z | z \in \mathbb{Z} \wedge z < 0\} \\
\gamma(\texttt{ 0- }) &= \{z | z \in \mathbb{Z} \wedge z \leq 0\} \\
\gamma(\texttt{ 0 }) &= \{0\} \\
\gamma(\texttt{ 0+ }) &= \{z | z \in \mathbb{Z} \wedge z \geq 0\} \\
\gamma(\texttt{ + }) &= \{z | z \in \mathbb{Z} \wedge z > 0\} \\
\gamma(\texttt{ +- }) &= \{z | z \in \mathbb{Z} \wedge z \neq 0\} \\
\gamma(\top) &= \mathbb{Z}
\end{aligned}$$

On the other hand, an element from the concrete domain is mapped onto the abstract domain with the **abstraction** function. This functions *abstracts* a property of the concrete element:

$$\alpha(n) = \left\{ \begin{array}{ll} \texttt{-} & \text{if } n < 0 \\ \texttt{0} & \text{if } n = 0 \\ \texttt{+} & \text{if } n > 0 \end{array} \right\}$$

The abstraction function α is used over sets as well throughout the text, the context should make clear which function is used. Abstraction of a set is equivalent to abstracting each element in the set, after which the union of all is taken:

$$\alpha(N) = \bigcup \{\alpha(n) | n \in N\}$$

These functions, both visualized in figure 4.3, give us a general way to deduce the behaviour of a function in the abstract domain. The abstract output for a function can be calculated by converting the abstract input to the concrete domain, using γ, resulting in a set. For this set, the concrete function is applied on each element, resulting in a new concrete set. This new concrete set is abstracted, giving the output for the abstract function:

$$\alpha(map(f, \gamma(input)))$$

This formula is unusable in a practical implementation. Calculating the function for *each* element of the conrete set is costly and often impossible. However, it offers an excellent way to derive the theoritical properties, offering a hint on how the practical implementation can be made.

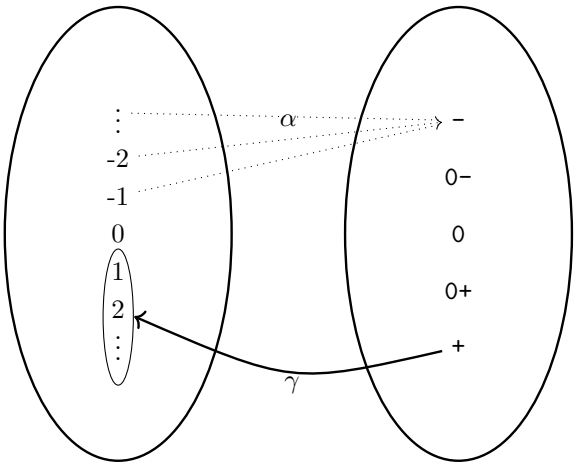


Figure 4.3: Concretization and abstraction between integers and signs

4.1.2 Ranges as abstract domain

Another possibility is to apply functions on an entire range at once (such as `[2,41]`), using abstract interpretation. Here, the abstract domain used are ranges of the form `[n, m]`.

This can be calculated by taking each element the range represents, applying the function on each element and abstracting the newly obtained set, thus

$$\alpha(map(f, \gamma(\texttt{ [n, m] })))$$

Where *map* applies f on each element of the set (like most functional programming languages), α is the abstraction function and γ is the concretization function, with following definition:

$$\begin{aligned} \alpha(n) &= \texttt{ [n, n]} \\ \alpha(N) &= \texttt{ [} min(N), max(N) \texttt{]} \\ \gamma(\texttt{ [n, m] }) &= \{x | n \leq x \leq m\} \end{aligned}$$

For example, the outputrange for `succ [2,41]` can be calculated in the following way:

$$\begin{aligned} & \alpha(map(\texttt{ succ }, \gamma(\texttt{ [2,41] }))) \\ = \; & \alpha(map(\texttt{ succ }, \{2, 3, ..., 40, 41\})) \\ = \; & \alpha(\{\texttt{ succ } 2, \texttt{ succ } 3, ..., \texttt{ succ } 40, \texttt{ succ } 41\}) \\ = \; & \alpha(\{3, 4, ..., 41, 42\}) \\ = \; & [3, 42] \end{aligned}$$

However, this is computationally expensive: aside from translating the set from and to the abstract domain, the function f has to be calculated $m - n$ times. By exploiting the underlying structure of ranges, calculating f should only be done *twice*, aside from never having to convert between the domains.

The key insight is that addition of two ranges is equivalent to addition of the borders:

$$[n, m] + [x, x] = [n + x, m + x]$$

Thus, applying `succ` to a range can be calculated as following, giving the same result in an efficient way:

$$\begin{aligned} & \texttt{succ [2,5]} \\ = \; & \texttt{[2,5]} + \alpha(1) \\ = \; & \texttt{[2,5]} + \texttt{[1,1]} \\ = \; & \texttt{[3,6]} \end{aligned}$$

4.1.3 Collecting semantics

As last example, the abstract domain might be the *set* of possible values, such as `{1,2,41}`. Applying `succ` to this set will yield a new set:

$$\begin{aligned} & \texttt{succ \{1,2,41\}} \\ = \; & \texttt{\{1,2,41\}} + \alpha(1) \\ = \; & \texttt{\{1,2,41\}} + \texttt{\{1\}} \\ = \; & \texttt{\{2,3,42\}} \end{aligned}$$

Translation from and to the abstract domain are trivially implemented. After all, the abstraction of a concrete value is the singleton containing the value itself, where the concretization

of a set of values is exactly the set of these values. This results in the following straightforward definitions:

$$\begin{array}{lcl} \alpha(n) & = & \{n\} \\ \gamma(\{n1, n2, \ldots\} & = & \{n1, n2, ...\} \end{array}$$

Performing the computation in the abstract domain of sets can be more efficient than the equivalent concrete computations, as the structure of the concrete domain can be exploited to use a more efficient representation in memory (such as ranges). Furthermore, different inputs might turn out to have the same result halfway in the calculation, such as `f x = abs(x) + 1` with input `{+1,-1}` which becomes `{1} + 1`. This state merging might result in additional speed increases.

Using this abstract domain effectively lifts a function over integers into a function over sets of integers. Exactly this abstract domain is used to lift the functions over parsetrees into functions over sets of parsetrees. To perform these calculations, an efficient representations of possible parsetrees will be constructed later in this section, in chapter 4.3.

4.1.4 Properties of α and γ

For abstract interpretation framework to work, the functions α and γ should obey to the properties *monotonicity* and *correctness*. These properties guarantee the soundness of the approach. The properties of the concretization and abstraction function also imply a **Galois connection** between the concrete and abstract domains.

Monotonicity of α and γ

The first requirement is that both *abstraction* and *concretization* are monotone. This states that, if the set to concretize grows, the set of possible properties *might* grow, but never shrink.

Analogously, if the set of properties grows, the set of concrete values represented by these properties might grow too.

$$\begin{array}{c} X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y) \\ X \subseteq Y \Rightarrow \alpha(X) \subseteq \alpha(Y) \end{array}$$

This is illustrated with the abstract domain of signs. Consider $X = 1, 2$ and $Y = 0, 1, 2$. This gives:

$$\begin{array}{lcll} & X \subseteq Y & \Rightarrow & \gamma(X) \subseteq \gamma(Y) \\ = & \{1,2\} \subseteq \{0,1,2\} & \Rightarrow & \alpha(\{1,2\}) \subseteq \alpha(\{0,1,2\}) \\ = & \{1,2\} \subseteq \{0,1,2\} & \Rightarrow & \texttt{+} \subseteq \texttt{0+} \end{array}$$

Per definition is `+` a subset of `0+`, so this example holds.

Soundness

When a concrete value n is translated into the abstract domain, $\alpha(n)$ should represents this value. An abstract object m represents a concrete value n iff its concretization contains this value:

$$\begin{array}{c} n \in \gamma(\alpha(n)) \\ \text{or equivalent} \\ X \subseteq \alpha(Y) \Rightarrow Y \subseteq \gamma(X) \end{array}$$

Inversly, some of the concrete objects in $\gamma(m)$ should exhibit the abstract property m:

$$m \in \alpha(\gamma(m))$$
or equivalent
$$Y \subseteq \gamma(X) \Rightarrow X \subseteq \alpha(Y)$$

This guarantees the *soundness* of the approach. This propery guarantees that the abstract object obtained by an abstract computation, indicates what a concrete computation might yield.

Without these properties tying α and γ together, abstract interpretation would be meaningless: the abstract computation would not be able to make statements about the concrete computations. For example, working with the abstract domain of signs where α maps `0` onto `+` yields following results:

$$\alpha(n) = \left\{ \begin{array}{ll} \texttt{-} & \text{if } n < 0 \\ \texttt{+} & \text{if } n = 0 \\ \texttt{+} & \text{if } n > 0 \end{array} \right\}$$

$$\gamma(\texttt{ + }) = \{n | n > 0\}$$
$$\gamma(\texttt{ - }) = \{n | n < 0\}$$

This breaks soundness, as $\gamma(\alpha(0)) = \gamma(\texttt{ + }) = \{1, 2, 3, ...\}$, clearly not containing the original concrete element 0. With these definitions, the approach becomes faulty. For example, `x - 1` with $x =$ `+` would become

$$\begin{array}{ll} & \alpha(\gamma(\texttt{ + }) - 1) \\ = & \alpha(\{n - 1 | n > 0\}) \\ = & \texttt{ +} \end{array}$$

A blatant lie, of course; `0 - 1` is all but a positive number.

Galois connection

Together, α and γ form a *Galois connection*, as it obeys its central property:

$$\alpha(a) \subseteq b \Leftrightarrow a \subseteq \gamma(b)$$

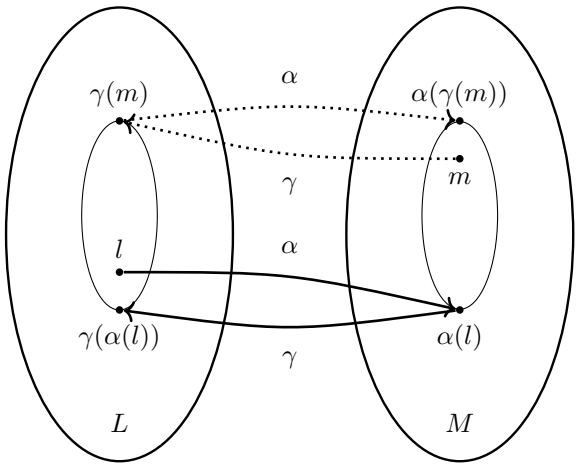


Figure 4.4: Galois-connection, visualized

4.2 Properties of the syntax

Abstract interpretation gives us a way to lift metafunctions over parsetrees to metafunctions over sets of parsetrees. To efficiently perform computations in this abstract domain, a representation for these sets should be constructed, exploiting the underlying structure of syntaxes. In this section, we study the structure and properties of the syntax, which are used in the next section to construct a set representation for parsetrees.

Some of these properties are inherent to each syntax, others should be enforced. For these, we present the necessary algorithms to detect these inconsistencies, both to help the programmer and to allow abstract interpretation.

4.2.1 Syntactic forms as sets

When a syntactic form is declared, this is equivalent to defining a set.

The declaration of `bool ::= "True" | "False"` is equivalent to declaring *bool* = { `True` , `False` }.

$$\texttt{bool} \longleftrightarrow \{\text{True, False}\}$$

This equivalence between syntactic forms and sets is the main driver for both the other properties studied and the efficient representation for sets introduced in section 4.2.

4.2.2 Embedded syntactic forms

Quite often, one syntactic form is defined in term of another syntactic form. This might be in a sequence (e.g. `int "+" int`) or as a bare choice (e.g. `... | int | ...`). If the latter case, each element of the choice is also embedded into the declared syntactic form.

In the following example, both `bool` and `int` are embedded into `expr`, visualized by ALGT in figure 4.5:

```
1 bool    ::= "True" | "False"
2 int     ::= Number       # Number is a builtin, parsing integers
3 expr    ::= bool | int
```

This effectively establishes a *supertype* relationship between the different syntactic forms. We can say that *every `bool` is a `expr`*, or `bool <: expr`.

This supertype relationship is a lattice - the absence of left recursion (see section 4.2.4) implies that no cycles can exist in this supertype relationship. This lattice can be visualized by ALGT, as in figure 4.6. Note that this lattice is cached in memory, allowing a lookup for the supertype relation.

4.2.3 Empty sets

The use of empty strings might lead to ambiguities of the syntax. When an emtpy string can be parsed, it is unclear whether this should be included in the parsetree. Therefore, it is not allowed.

As example, consider following syntax:

```
1 a       ::= "=" | ""
2 b       ::= "x" a "y" | "x" "y"
3 c       ::= a as
```

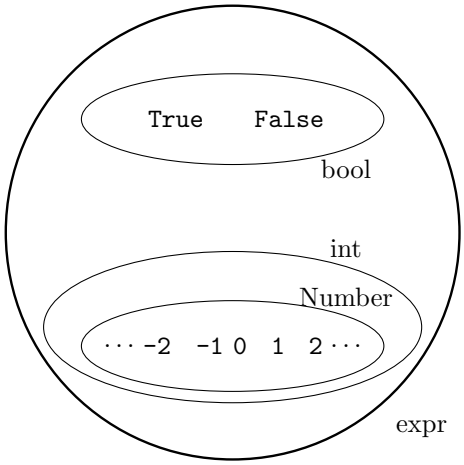


Figure 4.5: Nested syntactic forms

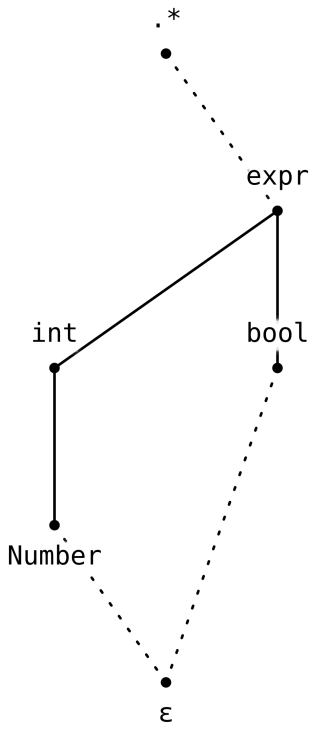


Figure 4.6: A simple subtyping relationship

Parsing `b` over string `x y` is ambiguous: the parser might return a parstree with or without an empty token representing `a`. Parsing `c` is even more troublesome, here the parser might return an infinite list containing only empty `a`-elements.

Empty syntactic forms can cause the same ambiguities, and are not allowed as well:

```
1 a       ::=             # empty, syntax error
```

While a syntactic form with no choices is a syntax error within ALGT, it is possible to define an empty form through recursion:

```
1 a       ::= a
2
3 a       ::= b
4 b       ::= a
```

Note that such an empty set is, by necessity, defined by using *left recursion*.

4.2.4 Left recursive grammers

A syntactic form is recursive if it is defined in terms of itself, allowing concise definitions of arbitrary depth. All practical programming languages do have grammers where syntactic forms are recursive. An example would be types:

```
1 type    ::= baseType "->" type | ...
```

Left recursion is when recursion occurs on a leftmost position in a sequence:

```
1 a ::= ... | a "b" | ...
```

While advanced parser-algorithms, such as *LALR-parsers* can handle this fine, it is not allowed in ALGT:

- First, this makes it easy to port a syntax created for ALGT to another parser toolchain - which possibly can't handle left recursion too.
- Second, this allows for a extremely simple parser implementation.
- Thirdly, this prevents having empty sets such as `a ::= a`.

Left recursion can be easily detected algorithmically. The algorithm itself can be found in figure 4.7. To make this algorithm more tangible, consider following syntax:

```
1 a       ::= "a" | "b" | "c" "d"
2 b       ::= a
3 c       ::= b | c "d"
```

First, the tail from each sequence is removed, e.g. sequence `"c" "d"` becomes `"c"`. This is expressed in lines 3-5 and has following result:

```
1 a       ::= "a" | "b" | "c"
2 b       ::= a
3 c       ::= b | d
4 d       ::= c
```

Now, all tokens, everything that is not a call to another syntactic form, is erased (lines 8-10):

```
1 a       ::=             # empty
2 b       ::= a
3 c       ::= b | d
4 d       ::= c
```

At this point, the main loop is entered: all empty syntactic forms and their calls are deleted (lines 16 till 21 for actual deletion):

```
1 b       ::=             # empty
2 c       ::= b | d
3 d       ::= c
```

In the next iteration, `b` is removed as well:

```
1 c       ::= d
2 d       ::= c
```

At this point, no syntactic forms can be removed anymore. Only syntactic forms containing left recursion remain[1], for which an error message can be generated (lines 24 till 26).

```
1  # For each sequence in each syntactic form,
2  # remove all but the first element
3  for each syntactic_form in syntax:
4          for each choice in syntactic_form:
5                  choice.remove(1..)
6
7  # Remove all concrete tokens (including builtins)
8  for each syntactic_form in syntax:
9          for each choice in syntactic_form:
10                 choice.removeTokens()
11
12
13 empty_rules = syntax.getEmptyRules()
14 while empty_rules.hasElements():
15         # remove each empty rule and occurences of it in other rules
16         for empty_rule in empty_rules:
17                 syntax.remove(empty_rule)
18                 for each syntactic_form in syntax:
19                         for each choice in syntactic_form:
20                                 choice.remove(empty_rule)
21         empty_rules = syntax.getEmptyRules
22
23
24 if syntax.isEmpty():
25         # all clear!
26 else:
27         error("Left recursion detected: "+syntax)
```

Figure 4.7: The algorithm to detect left recursion in a syntax

[1] Technically, syntactic forms containing a left recursive form on a leftmost position will be included too.

4.2.5 Uniqueness of sequences

When a parsetree is given, pinpointing exactly which syntactic form parsed it is usefull, as this can be used to minimize the set representation later on. A syntax should thus not contain duplciate sequences.

```
a ::= ... | "a" | ...
b ::= ... | "a" | ...

x ::= "x"
c ::= ... | a x | ...
d ::= ... | a x | ...
```

A parsetree containg `"a"` could be parsed with both `a` and `b`, which is undesired; the sequence `"a" "x"` could be parsed with both `c` and `d`. To detect this, we compare each sequences with each every other sequence for equality. When such duplicate sequences exist, the language designer is demanded to refactor this sequence into a new rule:

```
aToken  ::= "a"
a ::= ... | aToken | ...
b ::= ... | aToken | ...

x ::= "x"
ax ::= a x
c ::= ... | ax | ...
d ::= ... | ax | ...
```

This is not foolproof though. Some sequences might embed each other, as in following syntax:

```
a ::= "a"
b ::= a | "b"

c ::= "c"
d ::= c | "d"

x ::= a d
y ::= b c
```

Here, the string `"a c"` might be parsed with both syntactic forms `x` and `y`. There is no straightforward way to detect or refactor such a construction, without making things overly complicated. In other words, the approach refactoring duplicate constructions can not be used. Instead, runtime annotations are used to keep track of which form originated a parsetree.

The uniqueness-constraint is still added to keep things simpler and force the language designer to write a language with as little duplication as possible. Furthermore, it helps the typechecker to work more efficient and with less errors.

4.3 Representing sets of values

In this chapter, a general **representation** for a (possibly infinite) set of parsetrees is constructed. This representation is used as abstract domain for metafunctions, lifting any metafunction from a metafunction on parsetrees to a metafunction of sets of parsetrees. This set representation is constructed using the properties outlined in the previous chapter, using a small syntax as example:

```
1 baseType         ::= "Bool" | "Int"
2 typeTerm         ::= baseType | "(" type ")"
3 type             ::= typeTerm "->" type | typeTerm
```

4.3.1 Sets with concrete values

A set with only concrete values is simply represented by giving its inhabitants; the set `baseType` is thus represented as following:

```
1 { "Bool", "Int" }
```

We might also represent sequences of concrete values, in a similar way:

```
1 {"Bool" "->" "Bool"}
```

We could also create a set with, for example, all function types with one argument:

```
1 { "Bool" "->" "Bool"
2 , "Bool" "->" "Int"
3 , "Int" "->" "Bool"
4 , "Int" "->" "Int" }
```

4.3.2 Symbolic sets

A set can also be represented *symbolically*. For example, we might represent `baseType` also as:

```
1 { baseType } = { "Bool", "Int" }
```

While concrete values are written with double quotes around them, symbolic representations are not.

This symbolic representation can be used in a sequence too, with any number of concrete or symbolic values intermixed:

```
1   { baseType "->" baseType }
```

Which would be a succint notation for:

```
1 = { "Int" "->" baseType
2   , "Bool" "->" baseType}
3 = { "Bool" "->" "Bool"
4   , "Bool" "->" "Int"
5   , "Int" "->" "Bool"
6   , "Int" "->" "Int" }
```

In other words, this notation gives a compact representation of bigger sets, which could be exploited; such as applying a function on the entire set at once.

```
1 baseType         ::= "Bool" | "Int"
2 typeTerm         ::= baseType | "(" type ")"
3 type             ::= typeTerm "->" type | typeTerm
```

is translated to

```
1 baseType == {"Bool", "Int"}
2 typeTerm == {baseType, "(" type ")"}
3 type     == {typeTerm "->" type, typeTerm}
```

Figure 4.8: Translation of a syntax to set representation

4.3.3 Infinite sets

This symbolic representation gives rise to a natural way to represent infinite sets through inductive definitions, such as `typeTerm`:

```
1 type ::= { baseType, "(" type ")" }
2      = { "Bool", "Int", "(" typeTerm "->" type ")" , "(" typeTerm ")"}
3      = { "Bool", "Int", "(" "Bool" ")", "(" "Int" ")", ...
4      = ...
```

This notation allows abstract functions to run over infinite sets, another important feature of the abstract interpretation of metafunctions.

4.3.4 Set representation of a syntax

The BNF-notation of a syntax can be easily translated to this symbolic representation. Each choice in the BNF is translated into a sequence, rulecalls are translated into their symbolic value, closesly resembling the original definition. This can be seen in figure 4.8.

4.3.5 Conclusion

This representation of sets offer a compact representation of larger, arbitrary sets of parsetrees. As seen, the sets represented might even contain infinite elements. This representation could be the basis of many operations and algorithms, such as abstract interpretation. These algorithms are presented in the next section.

4.4 Operations on representations

In the previous chapter, an efficient and compact representation was introduced for sets of parse-trees - a big step towards multiple usefull algorithms and abstract interpreation of metafunctions.

However, constructing these algorithms requires basic operations to transform these sets. These operations are presented here.

4.4.1 Addition of sets

Addition is the merging of two set representations. This is implemented by simply taking all elements of both set representations and removing all the duplicate elements.

Per example, `{"Bool"} + {baseType}` yields `{"Bool", baseType}`. Note that `"Bool"` is embedded within basetype, the refolding operation (see section 4.4.3) will remove this.

4.4.2 Unfolding

The first important operation is unfolding a single level of the symbolic representation. This operation is usefull when introspecting the set, e.g. for applying pattern matches, calculating differences, ...

The unfolding of a set representation is done by unfolding each of the parts, parts which could be either a concrete value, a symbolic value or a sequence of those. Following paragraphs detail on how to unfold each part.

Unfolding concrete values

The unfold of a concrete value is just the concrete value itself:

```
1 unfold("Bool") = {"Bool"}
```

Unfolding symbolic values

Unfolding a symbolic value boils down to replacing it by its definition. This implies that the definition of each syntactic form should be known; making unfold a context-dependant operation. For simplicity, it is assumed that the syntax definition is passed to the unfold operation implicitly.

Some examples of unfolding a symbolic value would be:

```
1 unfold(baseType) = { "Bool", "Int" }
2 unfold(type)     = {(typeTerm "->" type), typeTerm}
```

Note the usage of parentheses around `(typeTerm "->" type")`. This groups the sequence together and is needed to prevent ambiguities later on. Such ambiguities might arise in specific syntaxes, such as a syntax containing following definition:

```
1 subtraction == {number "-" subtraction, number}
```

Unfolding subtraction two times would yield:

```
1 subtraction == { ..., number "-" number "-" number, ... }
```

This is ambiguous. The expression instance `3 - 2 - 1` could be parsed both as `(3 - 2) - 1` (which equals 0) and as `3 - (2 - 1)` (which equals 2). The syntax definition suggests the latter, as `subtraction` is defined in a right-associative way. To mirror this in the sequence representation, parentheses are needed:

```
1 subtraction == { ..., number "-" (number "-" number), ... }
```

Unfolding sequences

To unfold a sequence, each of the parts is unfolded. Combining the new sets is done by calculating the cartesian product:

```
unfold(baseType "->" baseType)
== unfold(baseType) × unfold("->") × unfold(baseType)
== {"Bool", "Int"} × {"->"} × {"Bool", "Int"}
== { "Bool" "->" "Bool"
   , "Bool" "->" "Int"
   , "Int" "->" "Bool"
   , "Int" "->" "Int" }
```

Unfolding set representions

Finally, a set representation is unfolded by unfolding each of the parts and collecting them in a new set:

```
unfold({baseType, baseType "->" baseType})
= unfold(baseType) + unfold(baseType "->" baseType)
= {"Bool", "Int"}
  + {"Bool" "->" "Bool"
    , "Bool" "->" "Int"
    , "Int" "->" "Bool"
    , "Int" "->" "Int" }
= { "Bool"
  , "Int"
  , "Bool" "->" "Bool"
  , "Bool" "->" "Int"
  , "Int" "->" "Bool"
  , "Int" "->" "Int" }
```

Directed unfolds

Throughout the text, unfolding of a set is often needed. However, an unfolded set can be big and unwieldy for this printed medium, especially when only a few elements of the set matter. In the examples we will thus only unfold the elements needed at hand and leave the others unchanged, thus using a *directed unfold*. It should be clear from context which elements are unfolded.

In the actual implementation, directed unfolds are sometimes used too. Depending on the context, the elements that should be unfolded are known. If not, refolding after an operation took place often has the same effect of a directed unfold.

4.4.3 Refolding

Refolding attempts to undo the folding process. While not strictly necessary, it allows for more compact representations throughout the algorithms - increasing speed - and a more compact output - increasing readability.

For example, refolding would change `{"Bool", "Int", "Bool" "->" "Int"}` into `{baseType, "Bool" "->" "Int"}`, but also `{type, typeTerm, "Bool"}` into `{type}`.

Refolding is done in two steps, repeated until no further folds can be made:

- Grouping subsets (e.g. `"Bool"` and `"Int"`) into their symbolic value (`baseType`)
- Filter away values that are embedded in another symbolic value (e.g. in `{"Bool", type}`, `"Bool"` can be omitted, as it is embedded in `"type"`)

This is repeated until no further changes are possible on the set. The entire algorithm can be found in figure 4.9, following paragraphs detail on each step in the main loop.

Grouping sets

Grouping tries to replace a part of the set representation by its symbolic representation, resulting in an equivalent set with a smaller representation (line 11 and 12 in the algorithm).

If the set defining a syntactic form is present in a set representation, the definition set can be folded into its symbolic value.

E.g. given the definition `baseType == {"Bool", "Int}`, we can make the following refold as each element of `baseType` is present:

```
1 refold({"Bool", "Int", "Bool" "->" "Int"})
2         = {baseType, "Bool" "->" "Int"}
```

Grouping sequences

Refolding elements within a sequence is not as straightforward. Consider following set:

```
1 { "Bool" "->" "Bool"
2 , "Bool" "->" "Int"
3 , "Int" "->" "Bool"
4 , "Int" "->" "Int" }
```

This set representation can be refolded, using the following steps:

- First, the sequences are sorted in buckets, where each sequence in the bucket only has a single different element (line 17):

```
1   ["Bool" "->" "Bool", "Bool" "->" "Int"]
2   ["Int" "->" "Bool", "Int" "->" "Int"]
```

- Then the different element in the sequences are grouped in a smaller set (line 20):

```
1   "Bool" "->" {"Bool", "Int"}
2   "Int" "->" {"Bool", "Int"}
```

- This smaller *difference set* is folded recursively (line 24):

```
1   "Bool" "->" refold({"Bool", "Int"})
2   "Int" "->" refold({"Bool", "Int"})
```

- This yields a new set; `{"Bool" "->" baseType, "Int" "->" baseType}`. As the folding algorithm tries to reach a fixpoint, grouping will be run again. This yields a new bucket, on which the steps could be repeated:

```
1   ["Bool" "->" baseType, "Int" "->" baseType]
2   -> {"Bool", "Int"} "->" baseType
3   -> unfold({"Bool", "Int"}) "->" baseType
4   -> baseType "->" baseType
```

This yields the expression we unfolded earlier: `baseType "->" baseType`.

Filtering embedded values

The second step in the algorithm is the removal of already represented values. Consider `{ baseType, "Bool"}`. As the definition of `baseType` includes `"Bool"`, it is unneeded in this representation.

This is straightforward, by comparing each value in the set against each other value and checking whether this element is embedded in the other (line 41 and 42).

```
refold(syntax, repr):
    do
        # Save the value to check if we got into a fixpoint
        old_repr = repr;

        # Simple, direct refolds (aka. grouping)
        for (name, definition) in syntax:
            # The definition set is a subset of the current set
            # Remove the definition set
            # replace it by the symbolic value
            if definition ⊆ repr:
                repr = repr - definition + {name}

        for sequence in repr:
            # a bucket is a set of sequences
            # with exactly one different element
            buckets = sortOnDifferences(repr)
            for (bucket, different_element_index) in buckets:
                # extract the differences of the sequence
                different_set = bucket.each(
                        get(different_element_index))

                # actually a (possibly smaller) set
                folded = refold(syntax, different_set)

                # Extract the identical parts of the bucket
                (prefix, postfix) = bucket.get(0)
                        .split(different_element_index)
                # Create a new set of sequences,
                # based on the smaller folded set
                sequences = prefix × folded × postfix

                # remove the old sequences,
                # add the new elements to the set
                repr = repr -  bucket + sequences

        for element in repr:
            for other_element in repr:
                if element == other_element:
                    continue
                if other_element.embeds(element):
                    repr = repr - element

    while(old_repr != repr)
    return repr
```

Figure 4.9: The algorithm to refold a set representation

4.4.4 Resolving to a syntactic form

Given a set, it can be usefull to calculate what syntactic form embeds all the elements in the set. E.g., each element from `{"Bool", "(" "Int" ")", "Int"}` is embedded in `typeTerm`.

This can be calculated quite easily:

- First, every sequence is substituted by the syntactic form which created it, its generator. In the earlier example, this would give:

```
1 | {baseType , typeTerm , baseType}
```

- The least common supertype of these syntactic forms gives the syntactic form embedding all. As noted in section 4.2.2, the supertype relationship of a syntax forms a lattice. Getting least common supertype thus becomes calculating the meet of these types (`typeTerm`).

4.4.5 Subtraction of sets

Subtraction of sets enabes a lot of usefull algorithms (such as liveability checks), but is quite complicated to implement.

Subtracting a set from another set is done by calculating the subtraction between all element of both sets. A single element, subtracted by another element, might result in no, one or multiple new elements.

There are a few different cases to consider, depending on what element is subtracted from what element. Following cases should be considered, each for which a paragraph will detail how subtraction is handled:

- A concrete value is subtracted from a sequence
- A symbolic value is subtracted from a sequence
- A sequence is subtracted from a symbolic value
- A sequence is subtracted from a sequence

In these paragraphs, the term **subtrahend** refers to the element that is *subtracted*, whereas the **minuend** refers to the element that is *subtracted from*. As mnemonic, the minuend will di*min*ish, whereas the subtrahend *subtracts*.

Subtraction of concrete values from a sequence

Subtraction of a concrete value from a concrete sequence only has an effect if the subtrahend and minuend are the same. Subtracting a concrete value from a different concrete value or subtracting it from a sequence has no effect.

Some example subtractions are:

- `"Bool" - "Bool"` is `{}` (thus the empty set)
- `"Int" - "Bool"` is `{"Int"}`
- `("Int" "->" "Int") - "Int"` is `{("Int" "->" "Int")}`

If the minuend is a single symbolic value, the subtraction only has an effect if the subtrahend is embedded within this symbolic value. If that is the case, the symbolic value is unfolded to a new set, from which the subtrahend is subtracted recursively. This can be seen in following examples:

- `baseType - "Bool"` equals `{"Bool", "Int"} - "Bool"`, resulting in `{"Int"}`
- `type - "("` equals `{type}`

Subtraction of a symbolic value from a sequence

Subtracting a symbolic value from a sequence could result in the empty set, when the subtrahend (the symbolic set) embeds the minuend:

- `"Bool" - baseType` is `{}`, as `"Bool"` is embedded in `baseType`
- `baseType - baseType` is `{}`, as `baseType` equals itself
- `"Bool" - type` is `{}` as `"Bool"` is embedded in `type` (via `typeTerm` and `baseType`)
- `baseType - type` is `{}` too, as it is embedded as well
- `("(" type ")") - type` is `{}`, as this is a sequence inside `typeTerm`

If the minuend is a single symbolic value which embeds the subtrahend, the resulting set is smaller than the minuend. Calculating the result is done by unfolding the minuend, from which the subtrahend is subtracted:

- `typeTerm - baseType` becomes `{baseType, "(" type ")"} - baseType`, resulting in `{"(" type ")"}`

Subtraction of a sequence from a symbolic value

If the minuend (a symbolic value) embeds the subtrahend (a sequence), then the result of the subtraction can be calculated by unfolding the minuend and then subtracting the subtrahend from each element, as can be seen in the following example:

```
1 ‘type - ("(" type ")")
2 = {typeTerm "->" type, typeTerm} - ("(" type ")")
3 = {typeTerm "->" type, baseType, "(" type ")"} - ("(" type ")")
4 = {typeTerm "->" type, baseType}
```

If the minuend (a symbolic value) does not embed the subtrahend, then the subtraction has no effect:

- `type - (expr "-" expr)` is `{type}`

Subtraction of a sequence from a sequence

The last case is subtracting a sequence from another sequence. Without losing generatlity, let the minuend to be the sequence `a b` and the subtrahend `a1 b1`, where `a` unfolds to `a1, a2, a3, ...` and `b` unfolds to `b1, b2, b3, ....` The sequence `a b` would thus unfold to the cartesian product, `a1 b1, a1 b2, a1 b3, ... , a2 b1, a2 b2, ...`

The subtraction would result in a set closely resembling the cartesian product, with only a single element gone, namely `a1 b1`. The resulting set, being `{ a2 b1, a3 b1, ...} + {a1 b2, a1 b2, ...} + {a1 b3, a2 b3, ...}` can be factored as `{(a - a1) ×b} + {a ×(b - b1)}`, which is a compact and tractable representation for this set.

This factorization is calculated by taking the pointwise differences between the sequences, after which the product with the rest of the sequence is taken. The pointwise difference is calculated by subtracting every ith subtrahend sequence from the ith minuend sequence (which implies both sequences have the same length).

Generalized to sequences from aribitrary length, the subtraction can be calculated by pointwise replacement of each element in the minuend:

```
1 a b c d ... - a1 b1 c1 d1  ...
2 = { (a - a1) b c d ...
3   , a (b - b1) c d ...
4   , a b (c - c1) d ...
5   , a b c (d - d1) ...
6   }
```

As example, the subtraction `(typeTerm "->" type) - ("Bool" "->" type)` yields following pointwise differences:

```
typeTerm - "Bool" = {"Int", "(" type ")"}
"->" - "->"       = {}
type - type       = {}
```

Resulting in the following set:

```
  { (typeTerm - "Bool") "->" type
  , typeTerm ("->" - "->") type
  , typeTerm "->" (type - type)}
= {{"Int", "(" type ")"} "->" type
  , typeTerm {} type
  , typeTerm "->" {}}
  # empty elements imply empty cartesian products
= {{"Int", "(" type ")"} "->" type}
= {"Int" "->" type, ("(" type ")") "->" type}
```

4.4.6 Conclusion

The set represention can be easily and efficiently transformed using the introduced operations, while still keeping the representation small. These operations form the basic building blocks for the further algorithms.

4.5 Lifting metafunctions to work over sets

In this chapter, the metafunctions are actually lifted to metafunctions over sets, using abstract interpretation. These metafunctions over sets will play a crucial role in later algorithms and the gradualization of programming languages.

The abstract interpretation framework, introduced in chapter 4.1 will guide the interpretation of patterns and expressions, the building blocks used to define functions; the set representation and operations introduced in the previous two chapters are used as data structure to make the computation feasible.

4.5.1 Translating metafunctions to the abstract domain

Metafunctions normally operate on parsetrees, which are elements of the concrete domain. In order to lift them to the abstract domain, sets of parsetrees, functions α (abstraction) and γ (concretization) are needed.

The abstraction-function is straightforward, as the smallest set representing a parsetree is the set containing only that parsetree.

Translating a set representation back to the concrete domain is done by enumerating all the elements represented in the set. The set representation, which possibly contains symbolic values, should thus be unfolded iteratively until only concrete values remain, reaching fixpoint for the unfold operation. The definitions of the abstraction and concretization functions can be found in figure 4.10.

This repeated unfolding of concretization is often impossible: a recursive definition (such as `type ::= baseType "->" type`) will result in an infinite set. The concretization function is only to study the properties of the abstract interpretation, but not for an actual implementation.

$$\begin{array}{rcl} \alpha(\texttt{ v }) & = & \{\texttt{v}\} \\ \gamma(w) & = & unfold^*(w) \end{array}$$

Figure 4.10: Definition of abstraction (α) and concretization (γ)

Monotonicity and Soundness

As seen in section 4.1.4, the functions α and γ should work in tandem to form a Galois-connection. If these functions would not form a Galois connection, the translations would mismatch and the approach breaks: the abstract computation might predict impossible computations.

The given functions do form a Galois-connection, as they fullfill monotonicity and soundness:

Lemma 1. *α (over sets) is monotone:*

$$\begin{array}{rcll} X \subseteq Y & \Rightarrow & \alpha(X) \subseteq \alpha(Y) & \\ & \Rightarrow & X \subseteq Y & \textit{(As } \alpha(X) = X\textit{)} \end{array}$$

Lemma 2. *γ is monotone:*

$$\begin{array}{rcll} X \subseteq Y & \Rightarrow & \gamma(X) \subseteq \gamma(Y) & \\ & \Rightarrow & unfold^*(X) \subseteq unfold^*(Y) & \textit{(Unfold does not change the contents of the represented set)} \\ & \Rightarrow & X \subseteq Y & \end{array}$$

Lemma 3. *α and γ are sound:*

$$\begin{array}{rcll} & & n \in \gamma(\alpha(n)) & \\ & = & n \in \gamma(\{n\}) & \textit{(Definition of } \alpha\textit{)} \\ & = & n \in unfold^*\{n\} & (\texttt{n}\ \textit{is a concrete value, thus can not be unfolded)} \\ & = & n \in \{n\} & \end{array}$$

and

$$\begin{array}{rcll} & & M \subseteq \alpha(\gamma(M)) & \textit{(α over a set returns that set)} \\ = & & M \subseteq \gamma(M) & \textit{(definition of } \gamma\textit{)} \\ = & & M \subseteq unfold^*(M) & \textit{(Unfolding a set representation has no influence on the represented set)} \\ = & & M \subseteq M & \end{array}$$

This implies that the set representation can actually be used as abstract domain for metafunctions.

4.5.2 Calculating a possible pattern match

Pattern matching can be interpreted in the domain of sets too - an important step in lifting metafunctions to the abstract domain. As pattern matching against a given pattern can be seen as a function from a parsetree to a variable store, the framework of abstract interpretation can be used to deduce the behaviour of pattern matching over sets.

There are three kinds of patterns that are considered:

- A pattern assigning to a variable

- A pattern comparing to a concrete value
- A pattern deconstructing the parse tree

In the following paragraphs, the behaviour of each of these patterns is explored. The symbol `~` is used to denote the pattern matching function, which takes a pattern and a parsetree to construct a variable store. Variable stores in the concrete domain will be denoted either `{Variable --> Parsetree}`, whereas variable stores in the abstract domain are denoted `{Variable --> {a, b, c}}` or `{Variable --> SomeSet}`.

As a variable store can be seen as a function from the variable name to the value, abstraction and concretization can be freely applied on the store, where $\alpha(\{$ `{ Var --> a }` , `{ Var --> b }` $\}) =$ `{ Var -->` α`a, b }` .

The second aspect of pattern matching is the aspect that a match can *fail*. This is modeled by, apart from the variable store, returning a set which *might* match. This function is denoted by `expr ~? Set`

Variable assignment

The first pattern to consider is variable assignment:

```
Var ~ Set
```

As each value in the set could be assigned in some concrete computation, the intuition for the abstract computation is that the entire set is assigned to the variable. This intuition is correct and can be formalized with the general formula of abstract interpretation, namely α`(map(f,` γ`(set)))`, where `f` is the pattern match against `Var`:

$$
\begin{aligned}
& \alpha(map(\ Var, \gamma(Set))) \\
= \ & \alpha(map(\ Var, unfold^*(Set))) \\
= \ & \alpha(map(\ Var, ..., a, b, c, ...)) \\
= \ & \alpha(..., \{\ \texttt{Var --> a}\ \}\ ,\ \{\ \texttt{Var --> b}\ \}\ ,\ \{\ \texttt{Var --> c}\ \}\ , ...\}) \\
= \ & \{\ \texttt{Var --> ..., a, b, c, ...}\ \} \\
= \ & \{\ \texttt{Var --> Set}\ \}
\end{aligned}
$$

Assignment to a variable can never fail, so the matching set is exactly the input set: `Var ~? Set = Set`.

The behaviour of the pattern store when the variable `Var` is matched twice is handled by the deconstruction pattern, as it is the responsibility of the deconstruction to merge multiple variable stores.

Concrete value

The second pattern is matching against a literal:

```
"Literal" ~ Set
```

This pattern does not add variables to the store, but only checks if the match might hold, returning an empty store for the match or failing if the concrete literal is not the same.

The behaviour over sets is thus mainly driven by the question of `"Literal"` is an element of the set or not:

$$
\begin{aligned}
& \alpha(map(\ \texttt{"Literal"}\ , \gamma(Set))) \\
= \ & \alpha(map(\ \texttt{"Literal"}\ , unfold^*(Set))) \\
= \ & \begin{cases} \alpha(map(\ \texttt{"Literal"}\ , \{..., a, b, c, \ \texttt{"Literal"}\ ...\})) & \texttt{"Literal"} \text{ is member of the set} \\ \alpha(map(\ \texttt{"Literal"}\ , \{..., a, b, c, ...\})) & \texttt{"Literal"} \text{ is not member of the set} \end{cases} \\
= \ & \begin{cases} \alpha(\{..., FAIL, FAIL, FAIL, \{\}...\})) & \texttt{"Literal"} \text{ is member of the set} \\ \alpha(\{..., FAIL, FAIL, FAIL, ...\})) & \texttt{"Literal"} \text{ is not member of the set} \end{cases} \\
= \ & \begin{cases} \{\} & \texttt{"Literal"} \text{ is member of the set} \\ FAIL & \texttt{"Literal"} \text{ is not member of the set} \end{cases}
\end{aligned}
$$

In other words, pattern matching a set is possible if this literal is an element of the set. If not, the match fails. The matching set is thus: `"Literal" ~? Set = {"Literal"} ∩Set`

Deconstructing pattern

The last fundamental pattern is a sequence deconstructing a parsetree:

```
x y ~ Set
```

The first aspect is whether the sequence might match. Intuitively, only sets containing an instance of the sequence could match the deconstruction. As each element of the deconstruction could be a subpattern, not every instance of the sequence will match, but only the instances having the right subparts. Luckily, the instances matching is exactly the cartesian product of the matching subparts.

$$
\begin{aligned}
& \alpha(map(\ \texttt{?}\ xy, \gamma(Set))) \\
= \ & \alpha(map(\ \texttt{?}\ xy, \{..., x' \times y', b, c, ...\})) \\
= \ & \alpha(\{..., xy\ \texttt{?}\ x1y1, xy\ \texttt{?}\ x1y2, xy\ \texttt{?}\ x2y1, ..., xy\ \texttt{?}\ b, xy\ \texttt{?}\ c, ...\}) \\
= \ & \alpha(\{..., xy\ \texttt{?}\ x1y2, xy\ \texttt{?}\ x1y2, xy\ \texttt{?}\ x2y1, ..., FAIL, FAIL, ...\}) \\
= \ & \alpha(\{..., (x\ \texttt{?}\ x1 \times y\ \texttt{?}\ y1), (x\ \texttt{?}\ x1 \times y\ \texttt{?}\ y2), (x\ \texttt{?}\ x2 \times y\ \texttt{?}\ y1), ...\}) \\
= \ & \alpha((x\ \texttt{?}\ x1 + x\ \texttt{?}\ x2 + ...) \times (y\ \texttt{?}\ y1 + y\ \texttt{?}\ y2 + ...)) \\
= \ & (x\ \texttt{?}\ x') \times (y\ \texttt{?}\ y')
\end{aligned}
$$

This suggests a very practical algorithm for abstract pattern matching:

- As sequences are supposed to be unique (section 4.2.5), this implies that only a single subset should be considered.
- For these sequences, pattern match the parts and combine their results

The store generated by the deconstructing pattern is the sum of all the stores. In the case that a single variable that is declared on multiple places. In the concrete domain, the merge of the variable store checks that the value is the same in both stores, failing otherwise. This implies that, in the abstract domain, it should be possible that both contain the same value; in other words, there should be a common subset which the variable might be.

$$
\begin{aligned}
& merge(\ \{\ \texttt{T --> Set1}\ \}\ ,\ \{\ \texttt{T --> Set2}\ \}\ ) \\
= \ & \{\ \texttt{T --> Set1 ∩ Set2}\ \}
\end{aligned}
$$

If the intersection is empty, this means that no concrete values could ever match this pattern, indicitating a dead clause. This also indicates that the earlier calculation of the matchin set

(`x ~ x' ×y ~ y'`) is too simplistic, but working out the full details would lead to far. A simpler way to calculate the matching set is to use the resulting variable store, and backfill the pattern as if it was an expression; this is introduced in the next section.

Multiple arguments

Functions with multiple arguments can be handled just as sequences are. Consider a function taking two arguments `f : a -> b -> c`. The two arguments can be considered a new syntactic form, `arg0 ::= a b`, underpinning the our intuition.

In other words, all functions can be considered functions with one input argument and one output argument.

4.5.3 Calculating possible return values of an expression

Given a set a variable might be, the set representation of a function can easily be calculated. As function expressions are sequences with either concrete values or variables, the translation to a representation is quickly made:

- A concrete value, e.g. `"Bool"` is represented by itself: `{"Bool"}`
- A variable is represented by the types it might assume. E.g. if `T1` can be `{"(" type ")", "Bool"}`, we use this set in the expression
- A function call is replaced by the syntactic form it might return. Typically, the signature is used to deduce this the return type, but some algorithms use more accurate set representations
- The set produced by a sequence is the cartesian product of the parts.

This gives us all the tools needed to lift single-clause functions to functions over sets.

4.5.4 Combining clauses

Combining multiple clauses is the last step in lifting arbitrary functions to functions over sets.

For each clause, the output of that clause can be calculated if the input set for that clause is known. The trick lies in the calculation of what each clause might get as input. Given the input set `I` for the function, the first clause will receive this entire input and matches a subset `I1` of the input. The second clause will receive the nonmatching part, thus `I - I1`, again matching a subset. This way, the input for each clause can be calculation, together with the subset of the input *not* matching any clause of the function, giving the **totality check** for free. Furhtermore, if a single clause never matches any input, a **dead clause** is detected, giving another check for free.

```
1 f       : a -> b
2 # Input I
3 f(x:x)  = ...   # Matches I1
4 # Input I - I1
5 f(y:y)  = ...
6 # Input I - I2
7 ...
8 # I - In is fallthrough and will never match
```

This gives all the pieces needed to lift functions over parsetrees into functions over sets, in a practical and computable way.

Per example, if one would like to know the result of the domain function over the set `{"Bool", "Bool" "->" type, ("Int") "->" "Bool"}`, this can be calculated. As reminder, the definition of `domain` can be found in figure 4.11). The actual calculation of the function happens clause per clause:

- Clause 1 matches nothing of the set, no element of the set has parentheses
- Clause 2 matches `("Int") "->" "Bool"`, returning `"Int"`, leaving `{"Bool", "Bool" "->" type}` for the next clause
- Clause 3 matches `"Bool" "->" type`, returning "Bool"
- `"Bool"` is never matched and falls through, giving no result

The sum of the calculation is `{"Bool", "Int"}` which can be refolded to `{baseType}'`.

```
1 dom                          : type -> typeTerm
2 dom("(" T1 ")")              = dom(T1)
3 dom(("(" T1 ")") "->" T2) = T1
4 dom(T1 "->" T2)              = T1
```

Figure 4.11: Definition of the domain function, as introduced in chapter 3.4.1.

4.6 Conclusion

The techniques presented allow metafunctions to be lifted to metafunctions over sets automatically. The framework for abstract interpretation gives a mathematical underpinning, whereas the compact representation and efficient operations ensure the practicallity and computablitiy, even in the face of infinite sets.

Not only is this a crucial step in the gradualization of the target language, quite some checks can be implemented warning the programmer for missing cases. Lifting functions to functions over sets is thus a powerfull technique that can be used for many means.

Chapter 5

Gradualization of STFL

In this chapter, gradualization itself is investigated. First, the broad framework of gradual automatic reasoning is painted, from which the advantages of gradual typing follow, together with a small and intuitive example of how gradual typing works in practice, to get some minimal feeling with gradual typing.

Then, using this example, the breakage of preservation is illustrated. This lack of preservation property is used to simplify the runtime proposed by the AGT-paper, yielding smaller and simpler requirements for the runtime.

With the runtime and typechecker requirements in mind, we gradualize STFL twice. The first gradualization is done intuitively and informal to give insight in the requirements, the second time ALGT is used to automate the process as much as possible.

5.1 Gradual languages

A programming language consists of expressions or statements. For these statements, extra information can be calculated using some automatic reasoning tool. Based on the result of this automatic reasoning, some statements could be forbidden, for example when the automatic reasoning deems the statement faulty.

A widely used instance of such automatic reasoning tool is a typechecker. This is the tool which reasons about what type an expression might return. Expressions which combine operators with incompatible operands are brought to the attention of the programmer - often by refusing to compile any further. A lot of programming languages use this technique, such as Haskell, Java, C, ...

Sometimes this automatic reasoning is too strict or too cumbersome. When the reasoning is too strict, advanced constructions with intricate properties might be forbidden while they could be perfectly valid - often the case when extra assumptions are known but can not be modeled within the tool. On the other hand, the automatic reasoning might be too cumbersome to deploy, e.g. when a quick-and-dirty solution is good enough.

Typical languages using the dynamic approach with type checking are Lisp, Prolog, Python, ... Applying an operator on incompatible operands results in a runtime crash.

Both the static and dynamic approach have clear benefits and drawbacks. Using and mixing both throughout the same program would be tremendously usefull: full automatic reasoning within the code for that difficult and intricate algorithm; dynamic behaviour in the glue between components, avoiding unnecessary boilerplate.

In languages with a static typesystem, gradualizing the typesystem means that some parts of the code are statically typechecked, whereas some are not. The compiler takes (the lack of) typeannotations as indication to use static or dynamic behaviour: where type annotations are given, the compiler performs the static checks and guarantees that no type errors can occur there. Places in the code without annotations[1] are treated dynamically, implying that crashes due to type errors can occur there. This also implies that some special care should be given to the borders between the static and dynamic behaviour, to prevent data of a wrong type to enter a statically typed part of the program and cause problems where least expected.

In a gradually typed language, the programmer chooses where and how much type annotations are given. This allows not only omit typing when not necessary, but it also allows the migration of codebases. Software projects often start their life as a small proof of concept in a dynamic language, with the intention of being rewritten in a static language when necessary - which'll never happen. The costs of rewriting the project are always deemed to high by management. Gradual typesystems solve this problem: the typing information can be added gradually over time in the codebase. It is possible to start fully dynamic, by not giving any type annotations and move to a totally staticly typed program by giving all type annotations possible.

5.1.1 Some gradualization of STFL

The essence of a gradual typechecker implementation is **optimism** regarding the missing type information: where typing information is omitted, the typechecker will optimistically assume that the types will work out at runtime. The typing rules are changed to reflect this optimism.

Naturally, before the typing rules can be updated to handle the unknown type, a representation for this missing info is needed. In practice is the missing type information often represented by using `?` as type (instead of `Bool`, `Int` or some function type).

An example expression that passes the gradual typecheck but can not be evaluated, is as following:

```
1 | (\ x : ? . x + 1) True
```

This example should pass the typechecker without problems: the argument `True` is compared to the type `?`. The typechecker optimistically assumes that the function will need a boolean in the end. The typecheck of the function body will use typecheck `x + 1` using `?` for the type of `x`, again optimistically assuming that it will turn out to be an integer at runtime.

Letting the previous example pass the typechecker requires some changes. For example, the rule which typechecks addition (see figure 5.1), should be modified to accomodate the missing type information. This missing information can only come from the terms (`n1` or `n2`) and is only be used to check that both terms are typed as `Int`, with the predicates `n1Type == "Int"` and `n2Type == "Int"`.

If on of these terms turns out to be some other type (such as `Bool`), a type error is reported. If no typing information is known, thus if `n1Type` would be `"?"`, then typechecker should be optimistic and pretend that `n1` is of type `"Int"`.

The rule `[TPlus]` can be gradualized by replacing `==` with an operator doing exactly the comparison above: the operator `~`, named *consistency*. Consistency behaves as `==`, except that it accepts unknown types as well. When confronted with `?`, it is optimistic and holds. The definition of consistency can be found in figure 5.2.

Swapping out equality `==` for consistency `~` in the typechecking rule will give us the gradual typing counterpart of `[TPlus]`:

[1] Another technique to deal with missing type annotations is *inferencing* types: the typechecker will calculate what type a certain expression has and silently add the annotations, making typechecking possible again. This is possible and usefull, but out of scope for this dissertation.

```
Γ ⊢ n1, n1Type Γ ⊢ n2, n2Type  n1Type == "Int" n2Type == "Int"
------------------------------------------------------------ [TPlus]
Γ ⊢ n1 "+" n2, "Int"
```

Figure 5.1: The typing rule for addition, repeated

```
T1 = T2 : type
--------------- [ConsBase]
T1 ~ T2


Ta1 ~ Ta2      Tb1 ~ Tb2
---------------------------------- [ConsArrow]
Ta1 "->" Tb1   ~   Ta2 "->" Tb2


----------      [ConsLeft]
"?" ~ T2


----------      [ConsRight]
T1 ~ "?"
```

Figure 5.2: Natural deduction rules defining consistency

```
Γ ⊢ n1, n1Type Γ ⊢ n2, n2Type  n1Type ~ "Int"  n2Type ~ "Int"
------------------------------------------------------------ [TPlus]
Γ ⊢ n1 "+" n2, "Int"
```

This gives some feel of how a gradual typesystem should behave, on an intuitive level. However, some more aspects should be considered as explained in the next part.

5.2 Breaking preservation

Using a gradual typesystem breaks one of the essential properties for languages, namely preservation (a definition recap can be found in figure 5.3).

Preservation breaks on programs containing type inconsistencies which are overlooked by the gradual typesystem. An expression such as `(\x : ? . x + 1) True` passes the gradual typecheck and the reduction becomes `True + 1`, for which no typing rule exists. `True + 1` can *not* be typed, resulting in preservation not holding anymore.

Progress still holds in this scheme. Any well typed expression is either canonical or can be reduced, because the typing rules mimic the reduction rules; for each typing rule, a respective reduction rule exists which guarantees progress. However, by using gradual typing, a reduction rule might construct a new program which contains a type error, as showcased above.

The reduction relation, constructed for the static language, does not have to deal with type errors at runtime; the preliminary typecheck and preservation properties guarantee that no malformed expression is ever passed into the reduction relation. At runtime, it is important to detect when preservation breaks, as that is the moment that program execution should be halted with an error. Another aspect that the reduction relation should guarantee is that no crashes can occur in statically typed parts of the program.

```
1
2   e0 :: T          e0 → e1
3  ------------------------ [Preservation]
4           e1 :: T
5
6
7
8            e0 :: T
9  -------------------------- [Progress]
10  e0:value     |      e0 → e1
```

Figure 5.3: The properties *Preservation* and *Progress*

5.3 Abstracting Gradual Typing

As seen in the previous intuitive approach, both the typechecker and the runtime should be modified: the typechecker should be able to handle the missing type information, the runtime should detect when type errors occur.

It turns out that a static typesystem and language can be converted algorithmically into a gradual typesystem and accompanying runtime. This result is presented by Ronald Garcia, Alison M. Clark and Éric Tanter in their paper *Abstracting Gradual Typing* [**?**].

The AGT approach is an interplay of multiple components:

- The staticly typed language with its typechecker, which acts as input.
- A gradualized typechecker, which is constructed algorithmically.
- A runtime with intrinsic terms, carrying evidence that a typing might be possible.

5.3.1 The proposed dynamic runtime

The AGT-paper presents a runtime, in which each expression carries its typing information with it (intrinsic terms). These intrinsic terms carry the type of the expression and some evidence that their typing *might* hold - some evidence that the missing type information could work out at runtime.

Reduction of an intrinsic term not only involves reducing the actual expression, but also evolving the typing information. The typing of the expression itself does not change, but the evidence that the typing might hold should evolve to some new evidence which proofs typing for the newly obtained term, or crash the program if the evidence can not be evolved due to a typing error that is discovered.

Furthermore, the paper presents a way to construct the algorithm calculating the initial evidence for an expression, called the *interior*.

5.3.2 Simplifying the runtime

While the mathematics to achieve this are impressive, they are somewhat overengineerd. Having an algorithm available to calculate the evidence for a term, implies that at each reduction step, the evidence can simply be recalculated instead of evolved. This makes the implementation easier, at the cost of elegance. However, this simplification can be taken further. The major goal of this evidence is proving that the typing (still) holds at a given moment in the execution. By removing the evolution of evidence, the purpose of constructing explicit evidence itself is gone. Instead of constructing evidence using interiors, the gradual typechecker itself can be run at each

reduction step. If this typechecker succeeds, the expression is valid and can be reduced at least once more. If the typechecker fails, a type error can be generated.

Note that the typechecker will only fail iff the expression can not be evaluated any further: when no reduction rule can be applied, due to a type error, no typing rule will be applicable as well - because the typechecker structure mimicks the reduction structure. So, when an expression should be reduced and it is unclear if the typing still holds, simply trying to reduce the expression suffices. If reduction fails, an error message should be provided - either the implicit error message generated by ALGT, or an explicit bottom element incorporated in the language.

The drawback of this approach is that the reduction relation should be constructed as a dynamic runtime from the start, incorporating dynamic features such as casts and eventually a type error element. To perform these casts, the implementor is free to chose between two possible implementations: carrying around type information in the term (having a slight runtime memory cost) or calculating the type of an expression when needed (which is free for simple call by value languages, as seen later).

At last, some care should be taken constructing the dynamic reduction relation that no untyped data slips into a typed part of the code. This can be done inserting casts when necessary.

In conclusion, to gradualize a programming language, the ingredients needed are:

- A programming language definition (the syntax)
- A reduction relation supporting a dynamic runtime (eventually implicitly)
- A static typechecker

Gradualizing these involves mainly modifying the typechecker. The syntax itself only needs a minimal modification.

5.4 Gradualizing the typesystem manually

5.4.1 Determining the scope of the unknown type information

The first aspect of gradualization, is choosing how broad unknown type information is. The gradual type system might allow `?` to represent any type (such as `"Bool"`, `"Int"` or a function type such as `"Bool -> Int"`). Another choice could be that `?` only represent concrete data, thus `"Bool"` or `"Int"`. This is something the language designer chooses in function of the goal of the language.

For the remainder of the text, the scope of unknown type information is every possible type (unless mentioned otherwise).

5.4.2 Representing dynamic types

The first aspect of gradualizing a programming language is allowing the typesystem to deal with missing type information.

Of course, the lack of typeinformation should be represented in the language itself. This lack of type information will be represented with the type `?` which will act as a new basetype in the language. As such, the syntax is amended:

```
1 | baseType        ::= "Bool" | "Int" | "?"
```

No other changes regarding the types are needed. The programmer might choose to represent type errors explicitly, using a new syntactic form. This issue is handled in the section 5.5.

5.4.3 Gradualizing the typing relationship

Gradualizing the typesystem boils down to ensuring that the typing rules can handle the missing type information.

When a closer look is taken to the rules, two categories are found:

- Rules deriving type information from a value without comparison, such as `[TNumber]`, `[TBool]`, `[Tparens]` and `[Tx]`. The first two rules will never encounter unknown type variables; the latter two rules might receive `?` as type of the subexpression, but pass it along without inspecting it. These rules can be left unchanged.
- Rules comparing types to other rules, which is always done using the relation `==`. Some special care should be taken when an operand uses a function, as can be seen in the rule `[Tapp]`, where `dom(T1)` is compared to `dom(T2)`.

This notion of type equality, inherently driven by the function `equate`, should be modified in order to gradualize the typechecker. Intuitively, seen the optimistic nature of gradual typecheckers, equality should pass if one of the parameters is the unknown type, or check recursively if the type passed is a function type. This gives rise to the *consistency* relationship as described in [**?**].

The function `equate` is renamed to `isConsistent` and changed to:

```
1 isConsistent             : type -> type -> type
2 isConsistent(T, T)       = T
3 isConsistent(?, T)       = T
4 isConsistent(T, ?)       = T
5 isConsistent(Ta1 "->" Ta2, Tb1 "->" Tb2)
6                          = isConsistent(Ta1, Tb1) "->" isConsistent(Ta2, Tb2)
```

Analogously, we rename `==` into `~`, a purely cosmetic change to reflect the change of meaning, given a new natural deduction rule:

```
1  isConsistent(T1, T2)
2 ---------------------- [ConsBase]
3        T1 ~ T2
```

At last, the two functions `dom` and `cod` should be gradualized; they should be able to handle the unknown type. Remember that the unknown type might represent either any other type, such as `"Bool"`, `"Int"`, `"Bool -> Int"`, `"Int" -> "Int"`, ... The result of `dom("?")` would thus be some grouping of `dom("Bool")`, `dom("Int")`, `dom("Bool -> Int")`, `dom("Int" -> "Int")`, ... The first two function calls, `dom("Bool")`, `dom("Int")`, are not defined and are thus irrelevant. The latter two, `dom("Bool -> Int")`, `dom("Int" -> "Int")`, return `"Bool"` and `"Int"` respectively. Broadly speaking, the domain of a function type could be every possible type - including a function type as well. This is what `dom("?")` should return: *every possible type*. Luckily, there exist a typeterm expressing exactly that within our gradual language: the unknown type `"?"`. The domain function (and analogously codomain function) can thus be gradualized by adding a single extra clause:

```
1 dom      : type -> type
2 dom("(" T1 ")")            = T1
3 dom(("(" T1 ")") "->" T2) = T1
4 dom(T1 "->" T2)            = T1
5 dom("?")                   = "?"
```

This approach has been formalized in the AGT-paper by Garcia et al. [**?**], and will be used in the next section, in which the typesystem is gradualized in a more automated manner. The gradual typesystem can be found in 5.4.

```
 n:number
--------------          [Tnumber]
 Γ ⊢ n, "Int"


 b:bool
---------------         [Tbool]
 Γ ⊢ b, "Bool"



--------------------    [Tx]
 Γ[ x ":" T ] ⊢ x, T



 Γ ⊢ e, T
-------------------     [TParens]
 Γ ⊢ "(" e ")", T


 Γ ⊢ e, T'      T' ~ T
----------------------- [TAscr]
 Γ ⊢ e "::" T, T'



 Γ ⊢ n1, Int1   Γ ⊢ n2, Int2    Int1 ~ "Int"    Int2 ~ "Int"
------------------------------------------------------------- [TPlus]
 Γ ⊢ n1 "+" n2, "Int"


 Γ ⊢ c, TCond   TCond ~ "Bool"          Γ ⊢ e1, T0      Γ ⊢ e2, T1      T0 ~ T1
-------------------------------------------------------------------------   [TIf]
 Γ ⊢ "If" c "Then" e1 "Else" e2, T0



 ((x ":" T1) "," Γ) ⊢ e, T2
---------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", T1 "->" T2



 Γ ⊢ e1, Tfunc  Γ ⊢ e2, Targ    Targ ~ dom(Tfunc)
-------------------------------------------------     [Tapp]
 Γ ⊢ e1 e2, cod(Tfunc)



 consistency(T1, T2)
--------------------- [Eq]
 T1 ~ T2
```

Figure 5.4: The gradualized typesystem for STFL

5.5 The dynamic runtime

As earlier noted, the runtime itself should support dynamic features.

The first feature is the behaviour when a type error is encountered. This could be explicitly coded as an extra syntactic form, allowing the error to be handled from within the programming language itself. The language designer might leave type errors implicit as well, as the ALGT-tool will print an error message when the reduction of the expression can not continue - due to a type error. This latter approach, of implicit type errors, is taken in this dissertation to keep the examples clean and simple.

The second feature that the runtime should factor is that types might be the unknown type. There are two rules in the dynamic runtime which explicitly deal with types: `[EvalAscr]` and `[EvalLamApp]`.

The rule `[EvalAscr]` performs a runtime typecheck; an expression of the form `e : T` is reduced to `e`, iff `e` is of type `T`. Guided by the optimism of the gradual approach, a cast to the unknown type should always pass; `e : ?` should thus be equivalent to `e`. Analogously, if the type of `e` can not be derived and turns out to be the unknown type, then the assertion should pass as well. Just as with the typechecker, it suffices to replace `==` with `~`. Type errors are still detected; if the expected type would be `"Bool"`, but the typing of `e` would yield `"Int"`, then the consistency relationship would be undefined, blocking progress and resulting in a crash.

The rule `[EvalLamApp]` applies lambda abstractions, by substituting the argument in the function body. Before application, the reduction rule checks that the argument is canonical (`arg:value`), checks type of the argument (`arg::TArg`) and compares this against the expected type (`TArg == TExp`). This check is important, as it enables an important property of gradual typing: *type errors can not occur in statically typed parts of the program.* Without this check, an argument with the wrong type might get in the function body, creating havoc in a place it is not expected.

The type comparison should still be gradualized, for which a few cases can be distinguished:

- The expected type is dynamic, in which case the runtime check is unneeded.
- The expected type is given and the argument is a simple value. The strict semantics will force the argument to be fully evaluated before the typecheck is performed, thus the type will be easily computable.
- The expected type is given and the argument is a lambda abstraction. If the full type of the lambda expression is known, the type comparsion should pickup type inconsistencies. If the type of this lambda expression contains some unknowns, type errors can not happen by simply calling this lambda abstraction, guaranteeing the type safety within the typed parts.

In all these cases, the optimistic consistency relation can be used to compare the types with each other.

The changes to the runtime system could thus be summarized as replacing the operator `==` with the consistency operator `~`. The adjusted typing rules can be found in 5.5.

```
 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:canon      arg :: TArg     TArg ~ TExp
------------------------------------------------------------ [EvalLamApp]
 ("(" "\\" var ":" TExp "." e ")") arg → !subs:e(var, arg, e)
```

Figure 5.5: The dynamic runtime, which can handle gradual types

```
 Gradualized
*************

 Syntax Changes
================

typeTerm        ::= ... | "?"

Rename type to gtype
Rename typeTerm to gtypeTerm

 Function Changes
==================

Rename equate to isConsistent

# Actual function chagnes are added in the next section

 Relation Changes
==================

Rename (==) to (~), pronounced as "is consistent"
```

Figure 5.6: The simple changes to STFL, denoting the addition of the unknown type and renaming changed entities to reflect their new meaning.

5.6 Automating gradualization

The gradualization, as given above, boils down to a few changes

- Adding the unknown type `"?"` to the syntax.
- Renaming some functions and relations to indicate their new meaning.
- Ensuring all functions handling types can deal with the unknown type.

The first two are but refactorings, which can be fully automated with ALGT: a file format exists to refactor elements of a language, which can be applied to the original language yielding the gradualized language.

Making sure that each function can handle the unknown type is automated as well by the ALGT-tool. Using the abstract interpretation framework and algorithms as introduced in chapter 4, the behaviour of a function over the unknown type can be calculated.

5.6.1 Refactoring

The differences between STFL and GTFL can be denoted into a `.langauge-changes` file. When both the original language and changes are passed into ALGT, the changes described in the changes-file are executed, which results in the gradual variant.

For each aspect of the original language definition (such as `Syntax`, `Functions`, `Relations`, ...), a section depicting *new* entities or a section giving `Changes` can be added to the language file.

The erlier changes to the syntax and renamings can be stated easily, as can be seen in figure 5.6.

```
 Possible results
------------------

type
```

Figure 5.7: Part of the analysis of function `dom`, which gives the possible outcomes for `dom ({ t:type })`. The full analysis output can be found in the appendixes.

```
 Possible results
------------------

(No results, all input crashes)
```

Figure 5.8: Part of the analysis of function `dom`, which gives the possible outcomes for `dom ({ baseType })`

5.6.2 Gradualizing domain and codomain

The unknown type `"?"` is a strange element within the typesystem, as the unknown type `"?"` represents *a set of types*. The unknown type is thus of a fundamental different nature in comparison with the normal types. As the unknown type behaves as a set of types, the behaviour of a function over `"?"` should behave the same as the abstract interpretation over the set that the unknown type represents, as noted in the paper by Garcia et al. [**?**].

Exactly these abstract functions over sets are available in ALGT, using the abstract interpretation framework that was introduced earlier. Using the flags `--ifa dom --iaa (t:type)`, the tool will perform an analysis of the `dom` function over the set of all types, giving an implementation of the gradual counterpart of `dom`. The result, as can be seen in figure 5.7, gives us, again `type`. This indicates what the gradual counterpart of `dom` should return when presented `"?"`: it returns `"?"`. The extra clause `dom("?") = "?"` is thus needed in the gradualization-file.

For the codomain function `cod`, the same procedure can be followed, obtaining the analogous result that `cod("?") = "?"`.

If we would choose `"?"` to only represent `{"Bool", "Int"}`, invoking ALGT with `--ifa dom --iaa (t:type)` tells us, as can be seen in figure 5.8, that no output is possible at all, indicating that `dom("?")` is not defined in this scenario.

5.6.3 Gradualizing equate

As `equate` has two arguments, reverse engineering the behaviour of the gradual counterpart of `equate` is not as easy, but ALGT can help this process too.

For starters, if one of the arguments is the unkown type, the behaviour of equate can be tested by passing the abstract arguments `(t:type),...`, where ... is the other argument for which the behaviour is needed. For example, the abstract arguments `(t:type),"Bool"` would yield `"Bool"`. But rather then knowing the behaviour of `equate("?", "Bool")`, the behaviour of `equate("?", someType)` is needed for the gradualization. Luckily, the analysis of `equate` contains some more usefull information. As can be seen in figure 5.9, the detailed analysis of clause 0 hints that, if clause 0 matches, that its output would strictly equal one of the input arguments. The result of `equate("?", t)` should thus equal `t` (and by the symmetry of the arguments, `equate(t, "?") = t`).

At last, we must deduce how the function behaves when the unkown type would occur somewhere as part of the first argument. ALGT can give a hint here too, by performing the abstract interpretation with arguments `t "->" t, t`. The analysis, as can be seen in 5.10, gives away that the result will have the form of a function type, but only if the second argument is a function type as well, and iff both the domain and codomain of both arguments are the same, indicating another missing clause: `equate(Ta1 "->" Ta2, Tb1 "->" Tb2) = equate(Ta1, Tb1) "->" equate(Ta2, Tb2)`.

The unkown type can not hide at other places, as indicated by the declaration of the syntactic form of `type`, so no other cases should be investigated.

```
 Analysis of equate : type -> type -> type
==========================================


     Analysis of clause 0
    ......................

    Clause:
      equate(T, T)             = T

    Possible inputs at this point:
    #  (type, type)


    Possible results:
    0   type(arg0) , type(arg0)         --> type(arg0)      : "type"

    This clause uses equality in the patterns and might not match. No arguments are




   Falthrough
  ------------

  (type, type)



   Possible results
  ------------------

  type
```

Figure 5.9: The full analysis of `equate` , with abstract arguments `type,type`

```
 Analysis of equate : type -> type -> type
==========================================

     Analysis of clause 0
    ......................

    Clause:
      equate(T, T)             = T

    Possible inputs at this point:
    #  ((typeTerm "->" type), type)


    Possible results:
      (typeTerm(arg0:0) "->" type(arg0:2)): type/0
    , (typeTerm(arg0:0) "->" type(arg0:2)): type/0
        --> (typeTerm(arg0:0) "->" type(arg0:2)): type/0 : "type"
```

Figure 5.10: The relevant parts of the analysis of `equate` , with abstract arguments `type "->" type,type`

5.6.4 Clause additions

To sum up, some clauses should be added to the functions; this can be done with the following statements in the `.language-changes`:

```
 Function Changes
==================

dom              : gtype -> gtypeTerm
...
dom("?")         = "?"

cod              : gtype -> gtype
...
cod("?")         = "?"

equate           : gtype -> gtype -> gtype
...
equate("?", t)   = t
equate(t, "?")   = t
equate(t11 "->" t12, t21 "->" t22)
                 = equate(t11, t21) "->" equate(t12, t22)

Rename equate to isConsistent
```

5.7 Conclusion

Using ALGT, gradualizing programming languages is simplified, as an implementation of the gradual function is availbe to experiment with. Converting the implementation to a succint and correct function declaration within ALGT still has to be done manually, requiring some intuition of the programer.

Chapter 6

Implementation of ALGT

In this chapter, the codebase of ALGT is explored. All important technical choices are given, making a reimplementation of core-ALGT possible: the representation of the syntax, functions and natural deduction rules; together with their usage: the parser and interpreters for functions and rules. The algorithms are accompanied with Haskell-snippets or pseudocode, so some familiarity with basic haskell is required for this chapter. These illustrate the algorithms, but are often simplified. Complications for additional features, often conceptually simple yet tremendously practical, are omitted.

This chapter does not cover the abstract interpretation; these algorithms are already explained in detail in chapter 4.

6.1 Representation and parsing of arbitrary syntax

The first aspect of any programming language is its syntax. ALGT allows to denote these explicitly, on which a parser can be based as can be seen in 3.3.2. Of course, the BNF has a representation within ALGT, which is given by the following construct:

```
1 -- Representation of a single BNF-expression
2 data BNF        = Literal String        -- Literally parse 'String'
3                 | BNFRuleCall Name      -- Parse the rule with the given name.
4                 | BNFSeq [BNF]          -- Sequence of parts
```

The most fundamental element here is the `Literal`, which is the terminal given in the string. Calling a non-terminal or builtin value is represented with `BNFRuleCall`. By using external definitions for the builtins, the main representation can be kept small and clean. At last, single elements can be glued together using `BNFSeq`.

All these expressions are bundled into a syntax:

```
1 {-Represents a syntax: the name of the rule + possible BNFS -}
2 data Syntax     = BNFRules
3                         { bnf           :: Map TypeName [BNF]
4                         , lattice       :: Lattice TypeName
5                         }
```

All the syntactic forms are saved into the dictionary `bnf`, which contains a mapping from the name of the non-terminal onto all possible choices for that syntactic form. As a syntacic form also is a type for the metafunctions, this dictionary also doubles as store for known types.

The second responsibility of the syntax is keeping track of the supertyping-relationship. The lattice-data structure keeps track of what type is a subtype of what other types and will play a major role in the typechecker.

6.1.1 Target language parsing

A syntax as above can be interpreted as a program, acting on an input string generating a parsetree. The parser, which uses the *Parsec*-package in the background, is constructed recursively as can be seen in figure 6.1.

The entry point for the parser is `parseNonTerminal`, which takes the name of the rule that should be parsed - together with the syntax itself. In this syntax, the relevant choices are searched. These choices are tried one by one in `parseChoices`. The actual parsing in `parseChoices` is delegated to `parsePart`, which does the actual interpretation: tokens (`Literal`) are parsed literally, sequences (`BNFSeq`) are parsed using `parsePart` recursively. If a non-terminal (`RuleCall`) is encountered, `parsePart` calls `parseNonTerminal`, closing the loop.

6.2 Target program representation

Target programs are represented as *parsetrees*. The data structure responsible is structured as following:

```
1 data ParseTreeA
2         = MLiteral      {ptaContents :: String}
3         | MInt          {ptaInt :: Int}
4         | PtSeq         {ptaPts :: [ParseTreeA]}
```

This implementation falls apart in the concrete values (`MLiteral` for strings and `MInt` for numbers) and a node to combine parts into longer sequences: `PtSeq`. `PtSeq` acts as node element in the parsetree, whereas the concrete values are the branches.

In the actual implementation more information is tracked in the parsetree, such as what syntactic form constructed the parsetree, the starting position of each token and its lengh. For clarity, this extra information is omitted in this text.

The string `(1 + 2)` -when parsed with `e` from STFL- is represented by the parsetree
`PtSeq [MLiteral "(", PtSeq [MLiteral "1", MLiteral "+", MLiteral "2"], MLiteral ")" ].`
Its graphical representation can be seen in figure 6.2

6.3 Metafunctions

Just as the syntax, the metafunctions are explicitly represented within ALGT. Remember that all metafunctions operate on a parsetree -a part of the target program. The representation of functions is thus closely linked with these parsetrees, but yet straightforward:

- The `Function` data structure contains the body of the function -multiple `Clauses`- and the type of the function (the types of the arguments and the type it'll return). These types are given explicitly in the source code, thus no inference is needed.
- A `Clause` contains the `Expression` returned and zero or more expressions which will performing the pattern matches.
- An `Expression` acts both as the pattern match representation and as the expression constructing a new parsetree as result of the function.

```
1
2  -- Seaches the given rule in the syntax, tries to parse it
3  parseNonTerminal        :: Syntax -> Name -> Parser ParseTree
4  parseNonTerminal syntax@(BNFRules syntForms _) ruleToParse wsModeParent
5   | ruleToParse `M.notMember` syntForms
6                  = fail $ ruleToParse++" is not defined in the given syntax"
7   | otherwise    = do    let choices     = syntForms ! ruleToParse
8                          parseChoice syntax ruleToParse choices
9
10
11
12
13 -- Search one of the choices that can be parsed
14 parseChoices    :: Syntax -> Name -> [BNF] -> Parser ParseTreeLi
15 parseChoices _ ruleToParse _ []
16         -- All choices depleted, can not parse anything
17         = fail $ "Could not parse expression of the form "++ruleToParse
18 parseChoices syntax ruleToParse  (choice:rest)
19         = try (parsePart syntax choice)
20            <|>  parseChoices syntax name  rest
21
22
23
24
25 -- Parses a single BNF-part
26 parsePart       :: Syntax -> BNF -> Parser ParseTree
27
28 -- Parse exactly the literal str
29 parsePart _ tp _ (Literal str)
30                 = do    string str
31                         return $ MLiteral str
32
33 -- Parse a rulecall...
34 parsePart syntax _  bnf@(BNFRuleCall ruleToParse)
35  | isBuiltin bnf
36                 = do    let parser      = getParserForBuiltin bnf
37                         parsedStr       <- parser
38                         return $ MLiteral parsedStr
39  | otherwise            -- call another non-terminal recursively
40                 = parseNonTerminal rules ruleToParse wsMode
41
42 -- degenerate case for sequences: a sequence with a single element
43 parsePart syntax (BNFSeq [bnf])
44                 = parsePart syntax tp  bnf
45
46 -- parse all elements of the sequence in order
47 parsePart syntax (BNFSeq (bnf:bnfs))
48                 = do    head    <- parsePart syntax tp  bnf
49                         tail    <- bnfs |+> parsePart' syntax tp
50                         return $ PtSeq (head:tail)
```

Figure 6.1: The recursively constructed parser. `MLiteral` and `PtSeq` are part of the target program representation and explained in 6.2

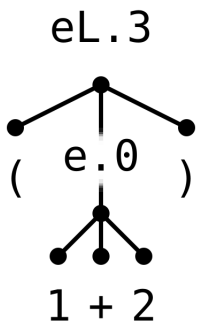


Figure 6.2: Parsetree of `(1 + 2)`

For each usage of an `Expression` (as detailed in section 3.4), an element is declared in the algebraic data type `Expression`:

- `MVar` is used to represent metavariables.
- `MParseTree` is used when a literal piece of target program is needed.
- `MCall` is a call to another function in scope.
- `MSeq` is a node in the meta-expression, grouping multiple parts to a bigger expression.
- `MAscription` will coerce the embedded expression to be of the given type.
- `MEvalContext` will, when used as pattern, create a hole within `Name` of the form `Expression`. When `MEvalContext` is used to construct a parsetree, it will plug the hole with the embedded expression.

Each `Expression` carries a `TypeName` as first argument, indicating the type of the expression. These types are inferred once the file is parsed.

```
1
2  -- A single pattern or expression
3  data Expression
4          = -- a variable
5            MVar TypeName Name
6            -- a 'value'
7          | MParseTree TypeName ParseTree
8            -- function call; not allowed in pattern matching
9          | MCall TypeName Name Builtin [Expression]
10           -- A node, containing multiple expressions
11         | MSeq MInfo [Expression]
12           -- checks wether the expression is built by this smaller type.
13         | MAscription TypeName Expression
14           -- describes a pattern that searches a context
15         | MEvalContext TypeName Name Expression
16
17 -- A single clause of the function body
18 data Clause
19         = MClause [Expression] Expression
20
21 -- A full function
22 data Function
23         = Function [TypeName] TypeName [MClause]
```

6.3.1 Typechecker of expressions

All expressions and patterns are typechecked, as type errors are easily made. Forcing parsetrees to be well-formed prevents the creation of strings which are not part of the language, what would result in hard to track bugs later on. Here, an overview of the inner workings of this typechecker (or more precisely _type annotator) are given. The pseudocode annotating expressions can be found in 6.5

These internals are simplified, as a type expection is always available: expressions and patterns are always typed explicitly, as the type signature of the function always gives a hint of what syntactic form[1] a pattern or expresion is. A type for a pattern indicates what type the pattern should deconstruct, or analogously for expressions, what syntactic form a parsetree would be if the expression was used to construct one. The natural deduction rules, which will be introduced in the following part, have the same typing available and can thus be typechecked with the same algorithm.

As expressions and patterns are **duals** in function of semantics, but the same in syntax and internal representation, the same typechecking algorithm can be used for patterns and expressions. However, some fundamental differences exist between in usage between patterns and expressions. The most striking example are variables: in a pattern context, an unknown variable occurence is a declaration; in an expression context, an unknown variable is an error In order to keep the typechecker uniform, the typechecker merely **annotates** types to each part of the expression; checks for unknown variables are done afterwards by walking the expression again.

To type **function calls**, a store γ containing all function signatures is provided. This dictionary γ is built before any typechecking happens by assuming the given function signatures are correct. A store for variables is not necessary, as variable typings are compared after the actual type annotation of expressions: a variable table is constructed, in which conflicts can be easily spotted.

With these prelimaniries, we present the actual typechecking algorithm used in ALGT. The algorithm has a number of cases, depending on the kind of expression that should be typed; composite expressions are handled by recursively typing the parts before handling the whole.

Variables

Variables are simply annotated with the expected type. One special case is when two (or more) patterns assign the same variable, such as the clause `f(x, x) = ...`. This is perfectly valid ALGT, as this clause will match when both arguments are identical. With the type signature `f : a -> b -> c` given, type of `x` can be deduced even more accuratly: the biggest common subtype of `a` and `b`, as `x` should be both an `a` and a `b`.

Actual type errors are checked after the initial step of type annotation, when all typing information is already available and inconsistencies can be easily detected.

To catch these inconsistencies between assignement and usage, the following strategy is used:

- First, pattern assignments are calculated; this is done by walking each pattern individually, noting which variables are assigned what types.
- When these individual pattern assignments are known, they are merged. Merging consists of building a bigger dictionary, containing all assigments. If two patterns assign the same variable, compatibility of the types is checked by taking the intersection of both types. If that intersection exists (a single common subtype), then some parsetrees exists which

[1] In this chapter, the term *type* is to be read as *the syntactic form a parsetree has*. It has nothing to do with the types defined in STFL. Types as defined within STFL will be denoted with `type`.

might match the pattern and this common subtype is taken as the type of the variable. If no such subtype exists, a type error is generated.
- With this store of all variable typings at hand, the expression can be checked for *undeclared* variables. This is simply done by getting the assignments of the expression -the same operation as on patterns- and checking that each variable of the expression occurs in the assigment of the patterns. If not, an unknown variable error is issued.
- The last step checks for inconsistencies between declaration and usage, which checks that a variable always fits its use, thus that no variable isused where a smaller type is expected.

The merging algorithm is listed in figure 6.6.

For example, the first clause of `f` (as seen in figure 6.3) has the typing assignemnts {"x" $\rightarrow$ { `bool` , `int` }}. As no intersection exists between `bool` and `int`, an error message is given.

```
1 not      : bool -> bool
2 not("True")      = "False"
3 not("False")     = "True"
4
5 f                : int -> bool -> ...
6 f(x, x)          = ...
```

Figure 6.3: Example of conflicting variable usage: `x` is used both as `bool` and `int` .

Sequences and string literals

As all expression have an explicit type expectation, typechecking expressions becomes easier. The possible sequences are gathered from the syntax definition and are aligned against the sequence to be typechecked. Then, each element is compared independantly: literals as given in the syntax definition should occor at the same position in the sequence, non-terminals should match the respective subexpressions, as can be seen in figure 6.4.

`"1"`	`"+"`	`x`	Should be typed as `expr`
`eL`	`"+"`	`e`	
`eL`	`"::"`	`type`	Literals don't match
`eL`	`e`		Not enough elements
`eL`			Not enough elements

Figure 6.4: An example sequence and possible typings

Functions

Functions are typed using the store containing all the function signatures. As all functions are explicitly typed, it is already available.

First, all the arguments are typed individually; then the return type of the function is compared against the expected type of the pattern/expression. The comparison used is, again, subtyping, as this always gives a sound result:

- When the funcions is used in an expression, a smaller type will fit nicely in the parsetree.
- When used as an pattern, the function is calculated and compared against the input parsetree. Here, the only requirement is that there exist *some* parsetrees that are common to

the argument type and the result type. In this case, the only check should be that some intersection of both types exists. As $expectedtype <: funtiontype$ gaurantees this, it is a sufficient condition. While this check is a little *to* strict, it is sufficient for practical use.

Type annotations

Type annotations are used for two means. First, it allows easy capturing of syntactic forms (e.g. `isBool((_:bool)) = "True"`). Secondly, they allow disambiguation of definitions in more complicated grammars.

A type annotations such as `x:T` is typechecked in two steps:

- The first check is that an expression of type `T` can occur at that location. This is easily checked: `T` should be a subtype of the expected type.
- The second check is that `x` can be typed as a `T`. This is done by running the type annotator recursively on `x`, with `T` as expectation.

Evaluation contexts

Evaluation contexts implement searching behaviour: when a parsetree is matched over `e[x]`, a subtree matching `x` is searched within the tree. If no such tree is found; the match fails. When this match is found, both `x` and `e` are available as variables.

The expression in the hole can be some advanced expression that should match the subtree. An other expression can be used in turn to construct a slightly different parsetree;

The explicit typing makes it possible to easily tag `e`, as its type `T` will already be stated by the function signature. However, it is difficult for the typechecker to figure out what type `x` might be. This is solved by typechecking `x` against *each* type that might occur (directly or indirectly) as subtree in `T`. If exactly one type matches, this typing is choosen. If not, an explicit typing is demanded.

This approach only works for complex expressions. Often, the programmer only wishes to capture the first occurence of a certain syntactic form, which can be written as `e[(b:bool)]`. In order to save the programmer this boilerplate, the typechecker attempts to discover a syntactic form name in the variable type. If this name is found (as prefix), it will be inherently typed. In other words `e[bool]` is equivalent to `e[(bool:bool)]`.

```
1
2  typecheckExpression(expr, γ, T):
3      case expr of:
4          variable v:     return v:T
5          sequence es:
6              # includes lone string literals, sequence of one
7              possible_typings = []
8              for choice_sequence in T.getChoices():
9                  if es.length != choice_sequence.length:
10                     continue
11                 try:
12                     typed_sequence = []
13                     for e, t in zip(es, choice_sequence):
14                         if e is literal && t is literal:
15                             if e != t then:
16                                 error "Inconsistent application
17                             else typed_sequence += e
18                         else:
19                             typed_sequence += typecheckExpression(e, γ, t)
20                     possible_typings += typed_sequence
21                 catch:
22                     # this doesn't match. Let's try the next choice...
23             if possible_typings == []:
24                 error
25                     "Could not match $expr against"
26                     "any choice of the corresponding syntactic form"
27             if possible_typings.length() > 1:
28                 error
29                     "Multiple possible typings for $expr."
30                     "Add an explicit type annotation"
31             return possible_typings[0]
32         function f(x1, x2, ...):
33             (T1, T2, ..., RT)    <- γ[f]    # Lookup type of f
34             x1'    = typecheckExpression(x1)
35             x2'    = typecheckExpression(x2)
36             if RT <: T:
37                 return f(x1', x2', ...) : RT
38             else:
39                 error
40                     "Function $f does not have the desired type"
41         type annotation (e:TA):
42             if !(TA <: T):
43                 error
44                      "The typing annotation is to broad"
45                      "or can never occur"
46             return typecheckExpression(e, γ, TA)
47         evaluation context e[x] with x a variable:
48             ts = T.occuringSubtypes().filter(x.isPrefixOf)
49             # occuringSubtypes are sorted on namelength
50             # the first match is the best match
51             t = ts[0]
52             typecheckExpression(e[(x:t)])
53         evaluation context e[x]:
54             ts = T.occuringSubtypes()
55             possible_typings = []
56             for t in ts:
57                 try{
58                 possibleTypings += typecheckExpression(x, γ, t)
59                 }catch():
60                     # This doesn't match. Let's try the next one
61             if possible_typings == []:
62                 error
63                     "Could not match $x against"
64                     "any possible embedded syntactic form"
65             if possible_typings.length() > 1:
66                 error
67                     "Multiple possible typings for $x."
68                     "Add an explicit type annotation"
69             return possible_typings[0]
```

Figure 6.5: The typechecking algorithm for meta-expressions and patterns

```
1  # Checks a clause for unknown or incompatible type variables
2  checkClause(pattern1, pattern2, ... , expr):
3      # search all the patterns for variables and their type
4      assgn1 = pattern1.assignedVars()
5      assgn2 = pattern2.assignedVars()
6      ...
7
8      assgnE = expr.assignedVars()
9
10     assgns = merge(assgn1, assgn2, ...)
11
12     for variable_name in assgnE.keys():
13         if !assgns.contains(variable_name):
14             error "Variable not defined"
15         TUsage = assgnE.get(variable_name)
16         TDecl = assgns.get(variable_name)
17         if !TUsage.isSubtypeOf(TDecl)):
18             error "Incompatible types"
19
20 merge(assgn1, assgn2, ...):
21     assgn    = {}
22     for variable_name in assgn1.keys() + assgn2.keys() + ... :
23         # assgn.get(T) return Top for an unknown type
24         type = assgn1.get(variable_name)
25                ∩ assgn2.get(variable_name)
26                ∩ ...
27         if type is ε:
28             error "Incompatible types while merging assignments"
29         assgn.put(variable_name, type)
30     return assgn
```

Figure 6.6: Merging of variable assignemnt stores and consistent variable usage checks

6.4 Interpretation of metafunctions

A function interpreter is builtin in ALGT, to evaluate the declared metafunctions. Just like any programming language, functions can be partial and fail. Failure can occur on two occasions:

- The builtin function `!error` is called
- A pattern match fails

6.4.1 Pattern matching

The input arguments of a function are matched against patterns for starters, simultanously dispatching input cases and building a variable store. The pattern matcher is defined by giving behaviour for each possible pattern, as can be seen in figure 6.7.

The recursive nature of the pattern matching is visible in the case of `MSeq` (line 6) where the input parsetree is broken down in pieces and analized further.

Another interesting detail is the function case (line 27): the function is evaluated using `evaluate func`. If the function fails and gives an error message, then the pattern match will fail automatically and control flow moves to the next clause in the function, providing a rudimentary error recovery procedure.

6.4.2 Parsetree construction

Given a variable store, an expression can be evaluated easily. Mirroring `patternMatch`, `evaluate` gives a parsetree for each expression as can be seen in figure 6.8.

```
1  patternMatch    :: Expression -> ParseTree -> Either String VariableAssignments
2  patternMatch (MVar v) expr
3                          = return $ M.singleton v (expr, Nothing)
4
5  patternMatch (MParseTree (MLiteral _ _ s1)) (MLiteral _ _ s2)
6   | s1 == s2             = return M.empty
7   | otherwise            = Left $ "Not the same literal"
8
9  patternMatch (MParseTree (MInt _ _ s1)) (MInt _ _ s2)
10  | s1 == s2             = return M.empty
11  | otherwise            =  Left $ "Not the same int"
12
13 patternMatch (MParseTree (PtSeq _ mi pts)) pt
14                         = patternMatch (MSeq mi (pts |> MParseTree)) pt
15
16 patternMatch  s1@(MSeq _ seq1) s2@(PtSeq _ _ seq2)
17  | length seq1 /= length seq2
18                         = Left $ "Sequence lengths are not the same"
19  | otherwise            = zip seq1 seq2 |+> uncurry (patternMatch )
20                                 >>= foldM mergeVars M.empty
21
22 patternMatch  (MAscription as expr') expr
23  | alwaysIsA (typeOf expr) as
24                         = patternMatch  expr' expr
25  | otherwise            = Left $ show expr ++" is not a "++show as
26
27 patternMatch func@MCall{} arg
28         = do    pt      <- evaluate func
29                 unless (pt == arg) $ Left $
30                         "Function result does not equal the given argument"
31                 return M.empty
32
33 patternMatch ctx _ pat expr
34         = Left $ "FT: Could not pattern match"
```

Figure 6.7: The pattern matching function. The code is edited for clarity, e.g. omitting a dictionary with known functions and the searching behaviour from evaluation contexts.

```
1  evaluate        :: VariableAssignments -> Expression -> Either String ParseTree
2  evaluate vars (MCall _ nm False args)
3   | nm `M.member` knownFunctions
4          = do    let func        = knownFunctions ! nm
5                  args'           <- args |+> evaluate ctx
6                  applyFunc ctx (nm, func) args'
7   | otherwise
8          = evalErr ctx $ "unknown function: "++nm
9
10 evaluate vars (MVar _ nm)
11  | nm `M.member` vars
12         = return $ fst $ vars ! nm
13  | otherwise
14         = Left $ "Unkown variable"
15
16 evaluate vars (MSeq tp vals)
17         = do    vals'   <- vals |+> evaluate vars
18                 return $ PtSeq vals'
19 evaluate vars (MParseTree pt)
20         = return pt
21 evaluate vars (MAscription tn expr)
22         = evaluate ctx expr
```

Figure 6.8: The parsetree construction, based on a meta-expression. The code is edited for clarity, e.g. omitting a dictionary with known functions, omitting evaluation contexts, ...

6.5 Relations

The last major feature of ALGT are the interpreter for relations. Per its declaration, has a relation one or more arguments, of which some input- and some output-arguments. The implementation of a relation consists of multiple natural deduction rules. These rules, reusing the patterns/expressions introduced for metafunctions, are represented internally with the following data type:

```
data Predicate  =
          -- Predicates as `x : type`
          TermIsA Name TypeName
          -- Predicates as `x --> y`
        | Needed RelationName [Expression]

data Rule       = Rule
        { -- The name of the rule, for documentation reasons only
          ruleName      :: Name
          -- The predicates that the rule should fullfill
        , rulePreds     :: [Predicate]
          -- The arguments to the relation it proves
        , ruleConcl     :: [Expression]
        }
```

With a representation for rules, relationships are represented as following:

```

data Relation = Relation
        { relationName  :: Name
        -- Input/output modes of the arguments, as declared
        , modes         :: [Mode]
        -- Types of the arguments, as declared
        , types         :: [TypeName]
        -- The rules implementing the relation
        , rules         :: [Rule]
        }
```

6.5.1 Typechecking

Typechecking relations is straightforward as well, the basic building block is already introduced in 6.3.1. As the type of each argument is already known, each expression in the conclusion of the rule can be annotated with types straightforwardly.

6.5.2 Building proof of a relation

With an overview of all rules and the relations that are implemented with them, it is possible to construct a proof that certain arguments $a_1, a_2, ...$ are part of a relationship R. Proving a relationship boils down to trying to proof any rule of the relationship with the given arguments; if such a rule is found, it is noted what rule is used. It might be possible that multiple rules provide a proof for some input argument, indiciting that a conclusion can be proven by multiple means. This is fine as long as the conclusion reached by the multiple rules is the same; but when the output arguments diverge, an error message is generated.

Proving a rule involves proving the predicates as well; this might involve proving another relation recursively.

The entire algorithm can be found as simplified pseudocode in figure 6.9

```
1
2  proofRelation(relationName, inputArguments):
3          possibleRules   = relations.rulesFor(relationName)
4          foundProofs     = []
5          for rule in possibleRules:
6                  proof   = proofRule(rule, inputArguments)
7                  if(proof.successfull()):
8                          foundProofs += proof
9
10         if(divergentConclusions(foundProofs)):
11                 fail "Divergent proofs"
12
13         # The shortest proof is selected
14         # This is merely cosmetically to keep the proof sizes as small as possible
15         return shortestProof(foundProofs)
16
17
18
19
20 # Proofs a single rule for the given arguments
21 proofRule(rule, inputArguments):
22         expressions     = rule.getExpressions()
23         predicates      = rule.getPredicates
24         inArgs  = zip inputArguments ruleExpressions.inputExpressions()
25         variableTable
26                 = patternMatchAll(inArgs)
27
28         # Proofs for the predicates
29         proofs  = []
30         for predicate in predicates:
31             # Proof each predicate in order; this might update the variableTable
32             (variableTable, proof) = proofPredicate(variableTable, predicate)
33             # If the predicate fails to be proven, then the rule proving fails too
34             if(!proof.successfull):
35                 fail "Predicate could not be proven"
36             proofs      += proof
37
38         return proofs
39
40
41
42 # Proof a single predicate
43 proofPredicate(variableTable, predicate):
44         case predicate of
45             # Proof a relation
46             # evaluate the arguments given and pass them to proofRelation
47             Needed relationName expressions:
48                 proofRelation(relationName,
49                         evaluateAll(expressions, variableTable))
50
51             # Check that the variable is of type type
52             TermIsA variable typename:
53                 if(variableTable.get(variable).type <: typename):
54                     return success
55                 else:
56                     fail "Incorrect type"
```

Figure 6.9: Pseudocode of the proof solving algorithm

```
1     e0 → e1
2 --------------- [EvalCtx]
3  e[e0] → e[e1]
```

Figure 6.10: A typical convergence rule, complicating the prover

6.5.3 Proving a relation with evaluation contexts

The searching behaviour of the evaluation context complicates the proving algorithm. To prove a typical convergence rule as `EvalCtx` (figure 6.10), a very specific `e0` is needed: one that can be reduced. This means that the pattern matching - where the subtree is searched - needs knowledge of the future predicates and should take these into account. The future predicates too are passed into the pattern matcher to achieve this. We omit the full details of this technicality, as that would take us to far, interested readers are refered to the source code available on github - ALGT is available on both the UGent and public repositories.

6.6 Further reading

The above code highlights the core features and algorithms of ALGT. This overview is far from complete: quite some practical features are omitted, apart from the many parts needed to make a usefull program (such as the language source parser, the command line parameter parser, the refactor support, ...)

Interested readers can find the entire codebase online, on `https://github.com/pietervdvn/ALGT/`.

Chapter 7

ALGT as tool for Language Design

In this concluding chapter, we reflect on ALGT and its position next to other tools in the language design community. We also reflect on the difficulties of gradual typing and the advancements made.

7.1 Context

Recall that two typing approaches exist in day-to-day programming: static and dynamic typing.

Static typing checks the program rigorously before executing the program for possible typing errors, whereas the dynamic approach crashes at runtime on such faulty expressions.

As is clear from the Tiobe Index [**?**], static programming languages are the most popular as the top 4 is filled with the static languages -*Java*, *C*, *C++* and *C#*. Nontheless, dynamic languages have their fair share in the top ten as well, as four spots are filled by the dynamic languages *Python*, *PHP*, *Javascript* and *Perl*.

Hybrid systems, such as gradual typing, are starting to emerge as these offer the benefits of both approaches. With gradual typing, programmers can choose, with a very fine grain, which parts of the code are statically or dynamically typed. On a larger timeframe can codebases be migrated from one approach to the other, depending on the needs of the program.

Some dynamic languages have an experimental, gradual typechecker available. Notable examples are MyPy for the Python language [**?**] or TypeScript, a superset with optional typechecker for Javascript [**?**]. Little is available, due to a lack of tools designing gradual programming languages.

7.1.1 Related work

As seen in section 2.2, some tools exist within the sphere of language design. Especially *PLT-redex* is a mature and advanced tool for language design, but has no support for gradualization. On the other hand, one highly specialized tool exists: *The Gradualizer* by Cimini. This tool gradualizes some languages, but is difficult to use due to lack of documentation and an arcane input method.

In conclusion, no tool exists which allows both the easy creation of a programming language and supports gradualization afterwards.

7.2 Main contribution

To fill this niche, a new tool ALGT was created. ALGT has a powerfull metalanguage for creating arbitrary programming languages. Secondly does ALGT help with the gradualization of a programming language.

7.2.1 Tool for language design

The first problem tackled by ALGT is the problem of easy and formal language design. The presented tool has a powerfull metalanguage, which allows all aspects of a programming language to be described in a high-level way. In order to evaluate the metalanguage, STFL was created within the tool, resulting in a clean and readable specification - as can be seen in chapter 3.

The syntax specification for a language can be given in the well-known *BNF* notation. The sytnax of STFL was constructed in BNF, as explained in section 3.3, resulting in a small and clean implementation.

This syntax definition is reused as data structure, on which transformations are based. These transformations can be given in two practical styles, one based on functions and one based on natural deduction.

The functional style consists of straightforward functions with little overhead. This functional approach is an excellent tool to create many smaller helper functions, as explained in 3.4.

The other style for transformations are in the form of relations, where the implementation is given by one or more natural deduction rules; as explained in 3.5. While having a little overhead, the natural deduction rules exhibit search behaviour making them an excellent tool to implement typecheckers or semantics.

Using this natural deduction style, semantics are given for STFL. Although any mathematical framework as described 1.1.2 is available, operational semantics were chosen over axiomatic or denotational semantics. Operational semantics are given for STFL in 3.5.5, again yielding a small and elegant specification.

A typechecker is constructed with the same tools in 3.5.6. Mirroring the operational semantics, the typechecker is small and elegant as well.

At last, important properties for the language can be stated and tested automatically as natural deduction rules. For STFL, this consisted of Progress and Preservation. Together with many of the other checks, this offers strong guarantees about the correctness of the designed language.

Based on the specification, an interpreter for the target language is fully automatically constructed. The BNF is enough to parse a target language creating an AST without needing any interaction of the language designer, as described in 6.1.1.

This AST is in turn transformed by a relation of choice. If a typechecking-relation is available, the target program can be typechecked; if semantics are availabe, the target program can be executed by proving that its reduction relation holds. The proof contains the execution trace and end result of the program as side effect, giving the output of the program.

With all these steps automated, language prototyping becomes easy. The two essential ingredients are declared easily and elegantly; using the automatic interpreter executing and testing the language is possible.

We tried to strike the right balance between the formal and lightweight tools.

- The language itself contains enough features to be practical, yet not too many, which might hinder formal proving of some properties.

- All the tools available are modeled after well-known mathematical objects, such as BNF to capture the syntax and natural deduction to capture semantics and typecheckers
- The semantics can be mechanically tested, increasing the formality and practicality of the tool
- A language specification is tested for many common bugs, by the typechecker, liveness- and completenesschecker, ...

A workshop was given to members of Zeus WPI, where uniniated programmers tried to create a programming language. Most were able to create the syntax of their language, but as none were familiar with natural deduction, implementation of the semantics was somewhat difficult.

In conclusion, ALGT is well suited for formal language development, with plenty tools to assist the designer: running and testing the language, catching bugs, ...

7.2.2 Tool for gradualization

The other major goal of ALGT is to assist and automate the gradualization of a typesystem. As seen in section 5.3.2, there are three components needed for gradualization:

- the language syntax with a dynamic type added;
- a runtime supporting dynamic features;
- a gradual typesystem.

All these components are constructed by hand for the earlier mentioned STFL in section 5.4.

Updating the syntax to allow a dynamic type is trivially done by adding a new choice `?` to the syntactic form.

The runtime which supports the dynamic features is still a task for the programmer. Luckily, STFL is already constructed to allow these features by having runtime typechecks available and already performing these casts at runtime when needed, as noticed in 5.5.

At last, the typesystem itself is gradualized. As it turns out in section 5.6.2, this boils down to converting some helper functions over types into functions over *sets of types*. To achieve this algorithmically, abtstract interpretation was used.

A framework for abstract interpretation over syntactic forms was constructed in chapter 4. This powerfull technique allowed to reason about an entire set passing through a function at once efficiently, based on an efficient notation for sets in combination with transformations over this notation.

This abstract interpretation gives guidance on how the gradualized functions should behave, allowing a gradual typesystem to be built. As bonus does the abstract interpretation give some extra checks for the metalanguage.

To test this abstract interpretation, gradual counterparts of `domain` and `codomain` were constructed in the process to gradualize STFL.

Gradualizing STFL manually is possible, altough some care should taken regarding implementation of the runtime and of the gradualized functions.

The next step is automating this gradualization. This is done by stating all the changes and refactorings needed for gradualization in a `.language-changes`-file; allowing easy recalculation of the gradual language if changes to STFL are performed. While gradualization is not fully automated, it allows both dialects to stay in sync.

7.3 Conclusion

All considered, ALGT is a powerfull tool which allows for easy and light, yet formally correct language design. The tool helps the designer, by offering a clutter-free experiences and helpfull error messages when small and bigger errors are made.

The tool also offers support to gradualize the typesystem.