Contents

Chapter 1

Creating STFL in ALGT

Now that the design goals are clear, 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, as to give a clear yet practical overview. When necessary, some key algorithms are presented (such as the typechecker). This chapter does *not* give an overview of the used command line arguments, 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`.

1.1 STFL

The *Simply Typed Functional Language* (STFL) is a small, functional language supporting integers, addition, booleans, if-then-else constructions and lambda expressions, as visible in the examples. Furthermore, it is strongly typed, with booleans, integers and higher-order functions.

STFL is the smallest language featuring 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`

1.2 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.

```
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
```

1.3 Syntax

The first step in formalizing a language is declaring a parser, which will turn the source code into a parsetree. In order to do so, we declare a context-free grammer, denoted as BNF, in order to construct a parser.

A context-free grammer can be used for two goals: the first is creating all possible string a language contains. On the other hand, we might use this grammer to deduce wether a given string is part of the language - and if it is, how this string is structured. This latter process is called *parsing*.

1.3.1 BNF

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

```
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 the section [syntax-properties]

1.3.2 STFL-syntax

In this format, the entire syntax for STFL can be formalized.

The first syntactic forms defined are types. Types are split into baseTypes and function types:

```
 Syntax
========

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

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

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

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

```
int     ::= Number
var     ::= Identifier
```

Expressions are split into terms and full expressions:

```

expr    ::= term "+" expr
        | term expr
        | term


term    ::= "If" expr "Then" expr "Else" expr
        | "(" "\\" var ":" type "." expr ")"
        | bool
        | int
```

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

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

1.3.3 Parsing

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

The parser in ALGT is a straightforward recursive descent parser, constructed dynamically by interpreting the syntax definition, of which the inner workings are detailed below.

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 will be 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. In order to do so, first `term` is parsed against the string. After inspecting all the choices for term, the parser will use the last choice of term (`int`), which neatly matches `22`. The string left is `+ 22`, which should be parsed against the rest of the sequence, `"+" expr`. The `"+"` in the sequence is now next to be parsed, which matches the head of the string. The last `22` is parsed against `expr`. In order to parser `22`, the last choice of `expr` is used.

A part of this process is denoted here, where the leftmost element of the stack is parsed:

```
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 ()
```

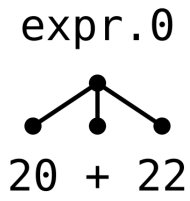

Figure 1.1: Parsetree of `20 + 22`

1.4 Metafunctions

Existing parsetrees can be modified or rewritten by using **metafunctions**[ˆtermFunction[. A metafunction receives one or more parsetrees as input and generates a new parsetree based on that.

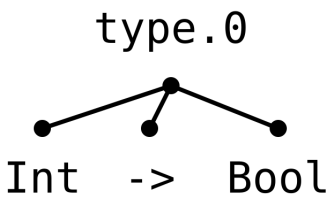

Figure 1.2: Parsetree of a function type

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 of the type[1] 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.
- Second, they assign variables, which can be used to construct a new parsetree.
- Third, they can deconstruct large parsetrees, allowing finegrained pattern matching withing the parsetrees branches.
- Fourth, advanced searching behaviour is implemented in the form of evaluation contexts.

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 seen 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.

[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`.

Expr	As pattern
`_`	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
`!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 subpart in it matching `pattern`
`a "b" (nested)`	Splits the parse tree 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
`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 parse tree

1.4.1 Domain and codomain

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

```
1 Functions
2 ===========
3
4
5 domain                          : type -> type
6 domain("(" T ")")               = domain(T)
7 domain(("(" T1 ")") "->" T2)    = T1
```

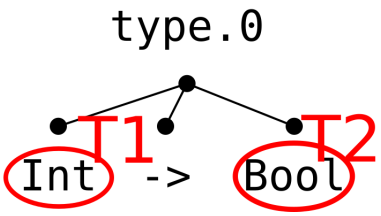

Figure 1.3: Pattern matching of `Int -> Bool`

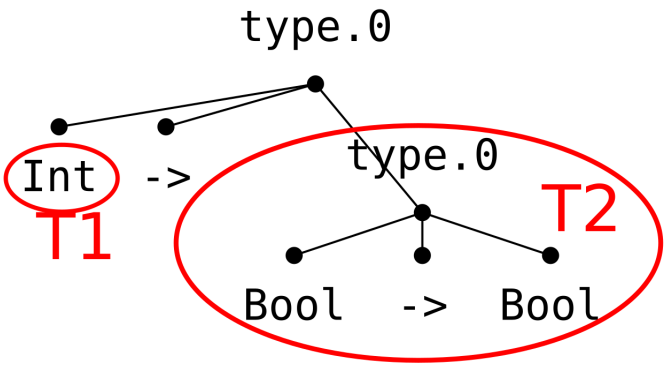

Figure 1.4: Pattern matching of `Int -> Bool -> Bool`

```
8  domain(T1 "->" T2)              = T1
9
10 codomain                        : type -> type
11 codomain("(" T ")")             = codomain(T)
12 codomain(T1"->" ("(" T2 ")"))   = T2
13 codomain(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 1.3. `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 1.4, again capturing the domain withing `T1`.

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 (`(Int -> Bool)`) and matching it against the embedded pattern `"(" T1 ")"`, as visualised in figure 1.5, capturing the domain *without parentheses* in `T1`.

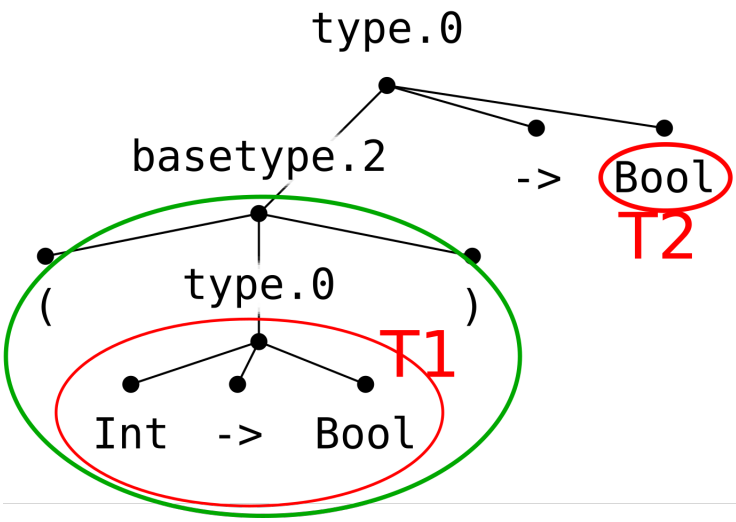

Figure 1.5: Pattern matching of `(Int -> Bool) -> Bool` . Matching in the subpattern is denoted with green

1.4.2 Typechecker

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 the typechecker are given.

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[2] 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,

[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`.

as variable typings are after the annotation.

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 infimum of both types. If that infimum is another syntactic form (another regular type), then some parsetrees exists which might match the pattern and the infimum type is taken as the type of the variable. If no such infimum exists (or more accuratly, if the infimum is bottom), a type error is detected.
- 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 is always fits its use, thus that no variable isused where a smaller type is expected.

This algorithm is listen in figure 1.7.

Sequences and string literals

Sequences (and also bare string literals) are handled by comparing them to the definition of the syntactic form, which consists of one or more BNF-sequences that can be chosen. Each of these defined sequences is tried by aligning it with the sequence that should be typed. Literal string values (and literal interger values) are filtered out here directly.

Functions

Functions are typed using the available function signature. 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 used in an expression, a smaller type will fit, thus subtyping is necessary. 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 the infimum of both types exists (more accuratly, that the infimum is not bottom). 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 capturign of syntactic forms (e.g. 'isBool((_:bool)) = "True") and it allows disambiguation of definitions in more complicated grammars.

In order to typecheck type annotations, the first issue is if that type *can* occur there. If the pattern `(_:expr)` is checked in a position where only a `bool` is a possible input argument, it's pretty useless to perform this annotation. The first check will thus be the sensibility of the annotation, namely that the annotated type `Ta` is a subtype of the expected type `T`.

If this check passes, the expression within the annotation is typed against the annotation.

Evaluation contexts

Evalaution contexts implement searching behaviour: when a parsetree is matched over `e[x]`, a subtree matching `x` -which can be a compound pattern- 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. `someExpr` can be used e.g. to extract information from a typing store, `e[someOtherExpr]` can be used to plug the hole with another value, e.g. to implement some form of substitution (although a builtin function is available for this).

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. In order to do so, `x` is typechecked as 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)]`.

Algorithm

All the pieces of the algorithm, as defined above, are put together in pseudocode in figure 1.6. Some trivial cases (such as pattern wildcard _) are omitted for clarity, despite their practical uses.

1.4.3 Minimal types, liveness- and totalitychecker

The functions are checked for various easily made errors, such as that each clause can match some input (liveness), that each input has a matching clause (totality) and that the declared output syntactic form is the smallest available.

These algorithms use of abstract interpretation and are thus detailed in a following chapter.

1.5 Operational semantics

1.6 Typechecker

1.7 Automatic checks

```
1
2  typecheck(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 += typecheck(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'    = typecheck(x1)
35             x2'    = typecheck(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 typecheck(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             typecheck(e[(x:t)])
53         evaluation context e[x]:
54             ts = T.occuringSubtypes()
55             possible_typings = []
56             for t in ts:
57                 try{
58                 possibleTypings += typecheck(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 1.6: The typechecking algorithm for meta-expressions and patterns

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

Figure 1.7: Merging of variable assignemnt stores and consistent variable usage checks