Chapter 1

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 **??**.

1.1 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 **??**. Of course, the BNF has a representation within ALGT, which is given by the following construct:

```
-- Representation of a single BNF-expression
data BNF        = Literal String        -- Literally parse ’String’
                | BNFRuleCall Name      -- Parse the rule with the given name.
                | 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:

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

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.

1.1.1 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 1.1.

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.

1.2 Target program representation

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

```
data ParseTreeA
        = MLiteral      {ptaContents :: String}
        | MInt          {ptaInt :: Int}
        | 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 1.2

1.3 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 `Clause`s- 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.

```

-- Seaches the given rule in the syntax, tries to parse it
parseNonTerminal        :: Syntax -> Name -> Parser ParseTree
parseNonTerminal syntax@(BNFRules syntForms _) ruleToParse wsModeParent
 | ruleToParse ‘M.notMember‘ syntForms
                = fail $ ruleToParse++" is not defined in the given syntax"
 | otherwise    = do    let choices     = syntForms ! ruleToParse
                        parseChoice syntax ruleToParse choices




-- Search one of the choices that can be parsed
parseChoices    :: Syntax -> Name -> [BNF] -> Parser ParseTreeLi
parseChoices _ ruleToParse _ []
        -- All choices depleted, can not parse anything
        = fail $ "Could not parse expression of the form "++ruleToParse
parseChoices syntax ruleToParse  (choice:rest)
        = try (parsePart syntax choice)
           <|>  parseChoices syntax name  rest




-- Parses a single BNF-part
parsePart       :: Syntax -> BNF -> Parser ParseTree

-- Parse exactly the literal str
parsePart _ tp _ (Literal str)
                = do    string str
                        return $ MLiteral str

-- Parse a rulecall...
parsePart syntax _  bnf@(BNFRuleCall ruleToParse)
 | isBuiltin bnf
                = do    let parser      = getParserForBuiltin bnf
                        parsedStr       <- parser
                        return $ MLiteral parsedStr
 | otherwise            -- call another non-terminal recursively
                = parseNonTerminal rules ruleToParse wsMode

-- degenerate case for sequences: a sequence with a single element
parsePart syntax (BNFSeq [bnf])
                = parsePart syntax tp  bnf

-- parse all elements of the sequence in order
parsePart syntax (BNFSeq (bnf:bnfs))
                = do    head    <- parsePart syntax tp  bnf
                        tail    <- bnfs |+> parsePart’ syntax tp
                        return $ PtSeq (head:tail)
```

Figure 1.1: The recursively constructed parser. `MLiteral` and `PtSeq` are part of the target program representation and explained in 1.2

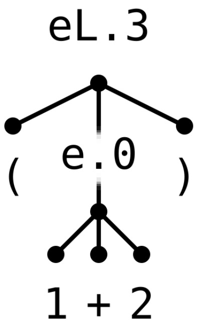

Figure 1.2: Parsetree of `(1 + 2)`

For each usage of an `Expression` (as detailed in section 1.3), 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.

```

-- A single pattern or expression
data Expression
        = -- a variable
          MVar TypeName Name
          -- a ’value’
        | MParseTree TypeName ParseTree
          -- function call; not allowed in pattern matching
        | MCall TypeName Name Builtin [Expression]
          -- A node, containing multiple expressions
        | MSeq MInfo [Expression]
          -- checks wether the expression is built by this smaller type.
        | MAscription TypeName Expression
          -- describes a pattern that searches a context
        | MEvalContext TypeName Name Expression

-- A single clause of the function body
data Clause
        = MClause [Expression] Expression

-- A full function
data Function
        = Function [TypeName] TypeName [MClause]
```

1.3.1 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 1.5

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[^syntacticType] 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 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 1.6.

For example, the first clause of `f` (as seen in figure 1.3) has the typing assignemnts {”x” $\to$ { `bool` , `int` }}. As no intersection exists between `bool` and `int`, an error message is given.

```
not      : bool -> bool
not("True")      = "False"
not("False")     = "True"

f                : int -> bool -> ...
f(x, x)          = ...
```

Figure 1.3: 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 1.4.

`"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 1.4: 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)]`.

```

typecheckExpression(expr, γ, T):
    case expr of:
        variable v:     return v:T
        sequence es:
            # includes lone string literals, sequence of one
            possible_typings = []
            for choice_sequence in T.getChoices():
                if es.length != choice_sequence.length:
                    continue
                try:
                    typed_sequence = []
                    for e, t in zip(es, choice_sequence):
                        if e is literal && t is literal:
                            if e != t then:
                                error "Inconsistent application
                            else typed_sequence += e
                        else:
                            typed_sequence += typecheckExpression(e, γ, t)
                    possible_typings += typed_sequence
                catch:
                    # this doesn't match. Let's try the next choice...
            if possible_typings == []:
                error
                    "Could not match $expr against"
                    "any choice of the corresponding syntactic form"
            if possible_typings.length() > 1:
                error
                    "Multiple possible typings for $expr."
                    "Add an explicit type annotation"
            return possible_typings[0]
        function f(x1, x2, ...):
            (T1, T2, ..., RT)    <- γ[f]    # Lookup type of f
            x1'    = typecheckExpression(x1)
            x2'    = typecheckExpression(x2)
            if RT <: T:
                return f(x1', x2', ...) : RT
            else:
                error
                    "Function $f does not have the desired type"
        type annotation (e:TA):
            if !(TA <: T):
                error
                     "The typing annotation is to broad"
                     "or can never occur"
            return typecheckExpression(e, γ, TA)
        evaluation context e[x] with x a variable:
            ts = T.occuringSubtypes().filter(x.isPrefixOf)
            # occuringSubtypes are sorted on namelength
            # the first match is the best match
            t = ts[0]
            typecheckExpression(e[(x:t)])
        evaluation context e[x]:
            ts = T.occuringSubtypes()
            possible_typings = []
            for t in ts:
                try{
                possibleTypings += typecheckExpression(x, γ, t)
                }catch():
                    # This doesn't match. Let's try the next one
            if possible_typings == []:
                error
                    "Could not match $x against"
                    "any possible embedded syntactic form"
            if possible_typings.length() > 1:
                error
                    "Multiple possible typings for $x."
                    "Add an explicit type annotation"
            return possible_typings[0]
```

Figure 1.5: The typechecking algorithm for meta-expressions and patterns

```
# Checks a clause for unknown or incompatible type variables
checkClause(pattern1, pattern2, ... , expr):
    # search all the patterns for variables and their type
    assgn1 = pattern1.assignedVars()
    assgn2 = pattern2.assignedVars()
    ...

    assgnE = expr.assignedVars()

    assgns = merge(assgn1, assgn2, ...)

    for variable_name in assgnE.keys():
        if !assgns.contains(variable_name):
            error "Variable not defined"
        TUsage = assgnE.get(variable_name)
        TDecl = assgns.get(variable_name)
        if !TUsage.isSubtypeOf(TDecl)):
            error "Incompatible types"

merge(assgn1, assgn2, ...):
    assgn    = {}
    for variable_name in assgn1.keys() + assgn2.keys() + ... :
        # assgn.get(T) return Top for an unknown type
        type = assgn1.get(variable_name)
               ∩ assgn2.get(variable_name)
               ∩ ...
        if type is ε:
            error "Incompatible types while merging assignments"
        assgn.put(variable_name, type)
    return assgn
```

Figure 1.6: Merging of variable assignemnt stores and consistent variable usage checks

1.4 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

1.4.1 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 1.7.

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.

1.4.2 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 1.8.

```
patternMatch    :: Expression -> ParseTree -> Either String VariableAssignments
patternMatch (MVar v) expr
                        = return $ M.singleton v (expr, Nothing)

patternMatch (MParseTree (MLiteral _ _ s1)) (MLiteral _ _ s2)
 | s1 == s2             = return M.empty
 | otherwise            = Left $ "Not the same literal"

patternMatch (MParseTree (MInt _ _ s1)) (MInt _ _ s2)
 | s1 == s2             = return M.empty
 | otherwise            =  Left $ "Not the same int"

patternMatch (MParseTree (PtSeq _ mi pts)) pt
                        = patternMatch (MSeq mi (pts |> MParseTree)) pt

patternMatch  s1@(MSeq _ seq1) s2@(PtSeq _ _ seq2)
 | length seq1 /= length seq2
                        = Left $ "Sequence lengths are not the same"
 | otherwise            = zip seq1 seq2 |+> uncurry (patternMatch )
                                >>= foldM mergeVars M.empty

patternMatch  (MAscription as expr') expr
 | alwaysIsA (typeOf expr) as
                        = patternMatch  expr' expr
 | otherwise            = Left $ show expr ++" is not a "++show as

patternMatch func@MCall{} arg
        = do    pt      <- evaluate func
                unless (pt == arg) $ Left $
                        "Function result does not equal the given argument"
                return M.empty

patternMatch ctx _ pat expr
        = Left $ "FT: Could not pattern match"
```

Figure 1.7: 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.

```
evaluate        :: VariableAssignments -> Expression -> Either String ParseTree
evaluate vars (MCall _ nm False args)
 | nm ‘M.member‘ knownFunctions
        = do    let func        = knownFunctions ! nm
                args’           <- args |+> evaluate ctx
                applyFunc ctx (nm, func) args’
 | otherwise
        = evalErr ctx $ "unknown function: "++nm

evaluate vars (MVar _ nm)
 | nm ‘M.member‘ vars
        = return $ fst $ vars ! nm
 | otherwise
        = Left $ "Unkown variable"

evaluate vars (MSeq tp vals)
        = do    vals’   <- vals |+> evaluate vars
                return $ PtSeq vals’
evaluate vars (MParseTree pt)
        = return pt
evaluate vars (MAscription tn expr)
        = evaluate ctx expr
```

Figure 1.8: 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, ...

1.5 Natural Deduction

Building on the pattern matching blocks of function interpretation, natural deduction takes the evaluation onto a next level.

1.5.1 Representation

1.5.2 Typechecking

1.5.3 Interpretation

1.6 Correctness proof for STFL

For completeness, we include a proof for preservation and progress of our definition of STFL.