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:

```
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.

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:

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

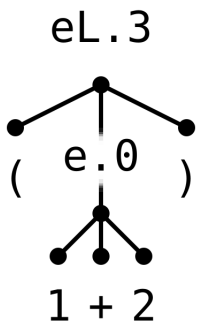

Figure 1.2: Parsetree of `(1 + 2)`

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.

```
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]
```

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[ˆsyntacticType] 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" $\rightarrow$ { `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 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)]`.

```
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 1.5: 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 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.

```
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 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 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]
        }
```

1.5.1 Typechecking

Typechecking relations is straightforward as well, the basic building block is already introduced in 1.3.1. As the type of each argument is already known, each expression in the conclusion of the rule can be annotated with types straightforwardly.

1.5.2 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 1.9

```
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 1.9: Pseudocode of the proof solving algorithm

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

Figure 1.10: A typical convergence rule, complicating the prover

1.5.3 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 1.10), 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.

1.6 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/`.

1.7 Correctness proof for STFL

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