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, 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.

1.1 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 1.1). 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 1.1: Example expressions of STFL and their end result

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. 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
```

1.3 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 wether 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.

1.3.1 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 the section about [Properties of the syntax].

1.3.2 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 typeTerms and function types:

```
typeTerm::= "Int" | "Bool" | "(" 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 and canonical form:

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

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

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

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

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

```
Resting string    stack
--------------    -----

"20 + 22"       ~ expr
"20 + 22"       ~ expr.0 (term "+" expr)
"20 + 22"       ~ term ; expr.0 ("+" expr)
"20 + 22"       ~ term.0 ("If" expr ...) ; expr.0 ("+" expr)
"20 + 22"       ~ term.1 ("(" "\" var ":" ...) ; expr.0 ("+" expr)
"20 + 22"       ~ term.2 (bool) ; expr.0 ("+" expr)
"20 + 22"       ~ bool ; term.2 (bool) ; expr.0 ("+" expr)
"20 + 22"       ~ bool.0 ("True") ; term.2 (bool) ; expr.0 ("+" expr)
"20 + 22"       ~ bool.1 ("False") ; term.2 (bool) ; expr.0 ("+" expr)
"20 + 22"       ~ term.3 (int) ; expr.0 ("+" expr)
"+ 22"          ~ expr.0 ("+" expr)
"22"            ~ expr.0 (expr)
"22"            ~ expr; expr.0 ()
...
"22"            ~ term.3 (int) ; expr.2 (); expr.0 ()
```

1.3.4 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

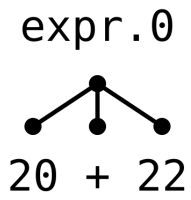


Figure 1.2: Parsetree of `20 + 22`

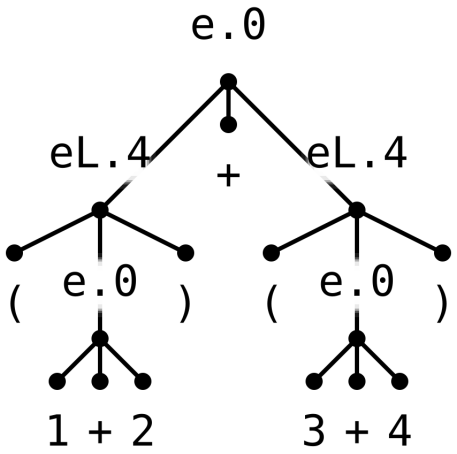


Figure 1.3: Parsetree of a `(1 + 2) + (3 + 4)` type

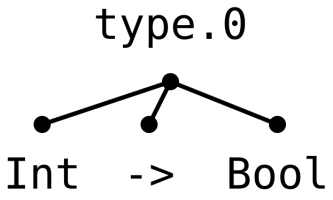


Figure 1.4: Parsetree of a function type

program file; resulting in the parsetree.

1.4 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 that. 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 structured way to do so. 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 explored in Convergence.

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:

[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
`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
`!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

1.4.1 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 Functions
```

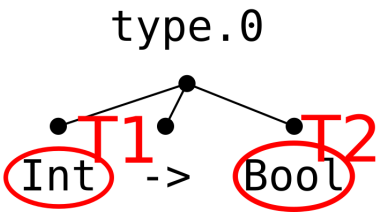


Figure 1.5: The variable assignment after pattern matching `Int -> Bool` against pattern `T1 "->" T2`

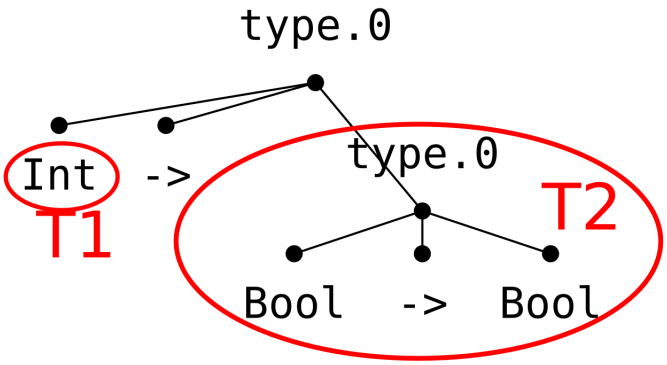


Figure 1.6: The variable assignment after pattern matching `Int -> Bool -> Bool` against pattern `T1 "->" T2`

```
2  ===========
3
4
5  domain                          : type -> type
6  domain("(" T ")")               = domain(T)
7  domain(("(" T1 ")") "->" T2)    = T1
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 1.5. `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 1.6, again capturing the domain within `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

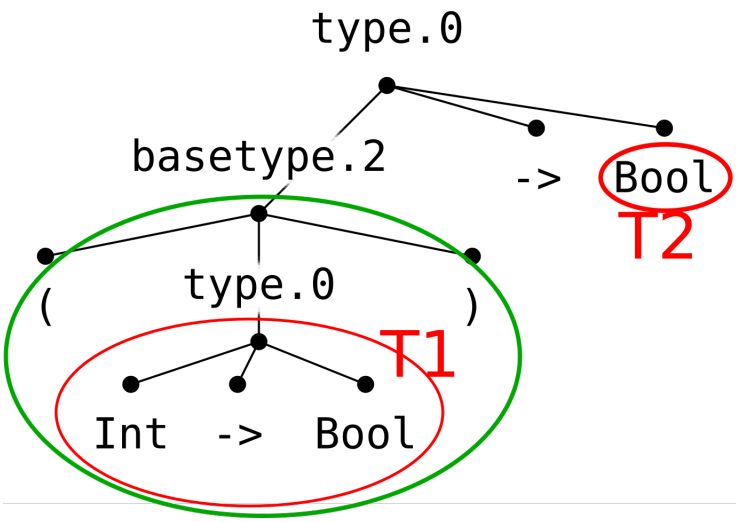


Figure 1.7: The variable assignment after pattern matching `(Int -> Bool) -> Bool` against pattern `("(" T1 ")") "->" T2`. The entire subpattern `"(" T1 ")"` is circled in green

part left of the arrow, namely `(Int -> Bool)`, and matching it against the embedded pattern `"(" T1 ")"`, as visualised in figure 1.7, capturing the domain *without parentheses* in `T1`.

1.4.2 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.

Readers interested in the actual typechecking of metafunctions, are referred to the chapter about internals; the checks catching liveness and fallthrough are explained in the chapter about abstract interpretation.

1.5 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.

1.5.1 Giving meaning to a program

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.

This transformation of the world can be achieved by many means:

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

These techniques all have their 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.

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.

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. 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.

1.5.2 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.

All these relations can be constructed using natural deduction rules in a straightforward and structured way. In the rest of this chapter, natural deduction is used to construct a smallstep semantic for STFL, followed by a typechecker.

1.5.3 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 → is a relation in `expr` ×`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.

1.5.4 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 1.9, where a rule introduced in figure 1.8 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 1.8: Example relation containing a single rule, which calls a metafunction

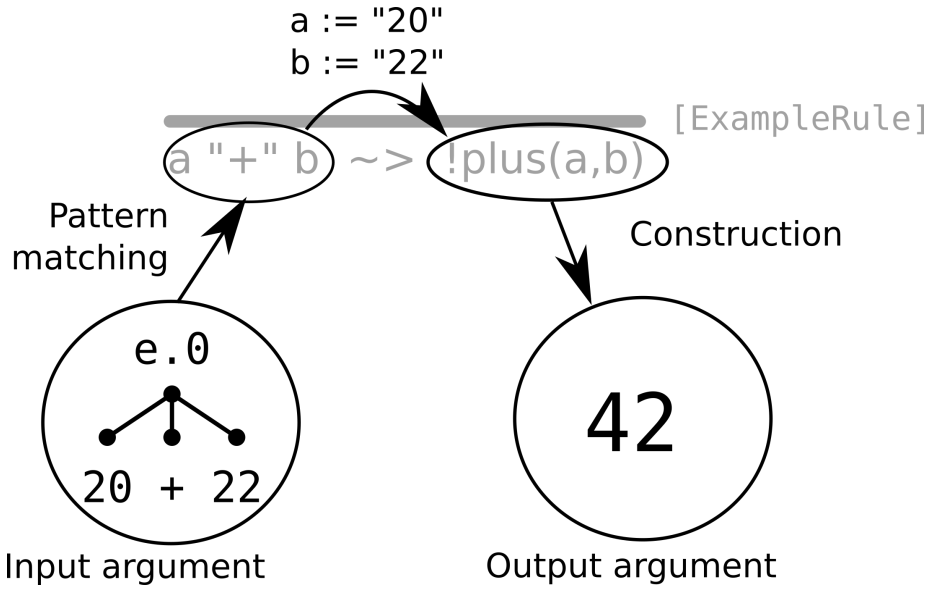


Figure 1.9: Application of the example rule, introduced in figure 1.8. The flow of information through this simple rule is denoted: the input argument is patterned matched, resulting in a variable store. Using this variable store, the output argument is constructed.

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 1.11.

```
 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 1.10: Example relation containing a predicate bounded rule, which is applied in 1.11

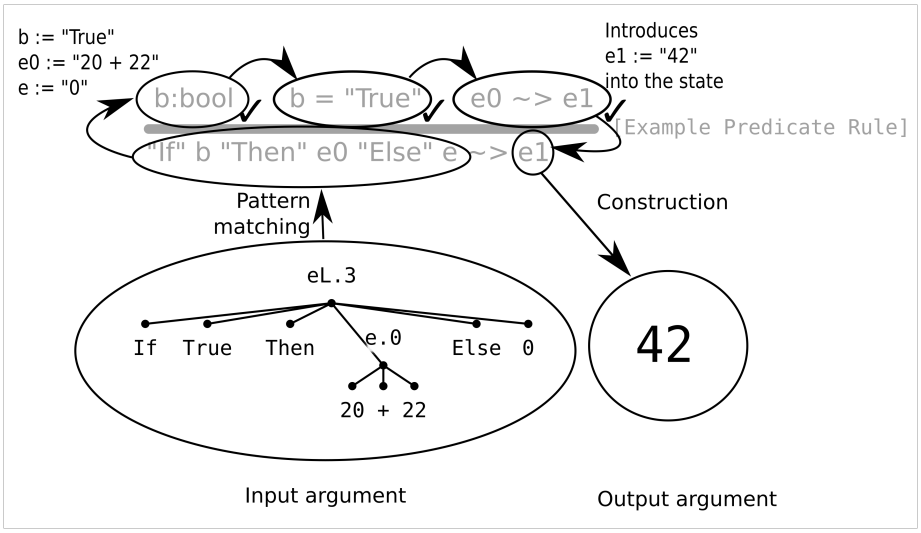


Figure 1.11: Application of a rule with an input argument, an output argument and predicates. The pattern matching of the input argument generates initial the state in the left upper corner, namely `b:="True", e0 := "20 + 22", e := "0"` . Each of the predicates is tested with this state in a left to right fashion. The last predicate, `e0 > e1` applies rule *ExampleRule* with `e0` as input argument and `e1` as unbound variable, as seen in 1.9. 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.

1.5.5 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. In order to do so, the builtin function `!plus` is used. 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:

```
 e  ::  T
------------------------                [EvalAscr]
 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:

```
 e :: T0         T == T0
------------------------                [EvalAscr]
 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:

```
 arg:value      arg :: T
------------------------------------------------ [EvalLamApp]
 ("(" "\\" var ":" T "." 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`.

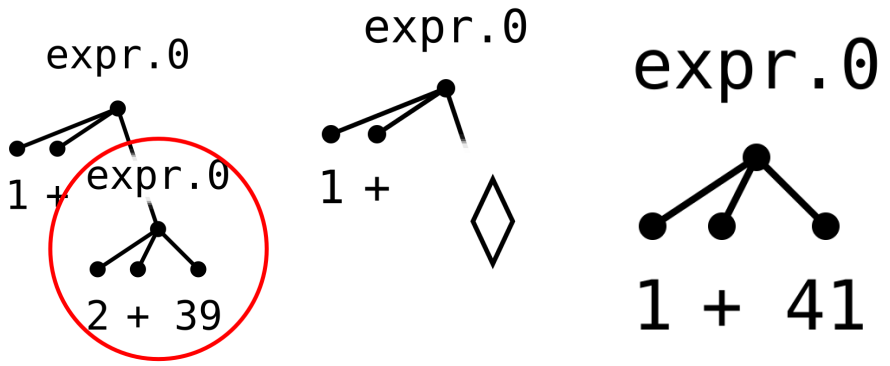
It is rather cumbersome to introduce rules for each position where a syntactic form might be reduced. For `+`, this would need two extra rules (`e0 "+" e →e1 "+" e` and `e "+" e0 →e "+" e1`), `If`-expressions would need an additional three rules, ... This clearly does not scale.

A scaling solution is the following convergence rule:

```
 e0 → e1
-----------------                       [EvalCtx]
 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 1.12.



(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 1.12: 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:value      arg :: T
35 ------------------------------------------------ [EvalLamApp]
36  ("(" "\\" var ":" T "." 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: 5, proof depth: 4


              41 : number
              ------------ [Tnumber]
              {} ⊢ 41, Int
              ---------------------- [TEmptyCtx]
41 : value    41 :: Int
---------------------------------- [EvalLamApp]
( \ x : Int . x + 1 ) 41 → 41 + 1
```

1.5.6 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*

1.5.7 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"
```

1.5.8 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 `==`, as this relation will be replaced by `~` in the gradualization step. `==` is defined in a straightforward way and can be replaced by `=` if needed.

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:

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

```
 Γ ⊢ c, "Bool"  Γ ⊢ e1, T0      Γ ⊢ e2, T1      T0 == T1
------------------------------------------------------        [TIf]
 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:

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

```
--------------------    [Tx]
 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:

```
 ((x ":" T1) "," Γ) ⊢ e, T2
------------------------------------------------------- [TLambda]
 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)
```

1.5.9 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.

1.5.10 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, "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)
```

1.5.11 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.

1.6 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.

1.6.1 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.

1.6.2 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 wether 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
```

1.6.3 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
```

1.6.4 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 +
  # 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]

1.7 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.