Contents

Chapter 1

Creating STFL in ALGT

Now that the design goals are clear, we present **ALGT**, a tool and language to describe both syntax and semantics of aribitrary programming languages. This can be used to formally capture meaning of any language, formalizing them or, perhaps, gradualize them.

To introduce the ALGT-language, a small functional language (STFL) is formalized, as to give a clear yet practical overview. When necessary, some key algorithms are presented (such as the typechecker). This chapter does *not* give an overview of the used command line arguments, exhaustive overview of builtin functions, ... For this, we refer the reader to the tutorial and manual, which can be obtained by running `ALGT --manual-pdf`.

1.1 STFL

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

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

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

1.2 Skeleton

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

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

Syntax
======

# Syntax definitions

Functions
=========

# Rewriting rules, small helper functions

Relations
=========

# Declarations of which symbols are used

Rules
=====

# Natural deduction rules, defining operational semantics or typechecker

Properties
==========

# Automaticly checked properties
```

1.3 Syntax

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

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

1.3.1 BNF

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

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

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

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

A language containing all integers has already been provided via a builtin:

```
int             ::= Number
```

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

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

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

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

Note that some syntactic forms are not allowed (such as empty syntactic forms or left recursive forms), this is more detailed in the section [syntax-properties]

1.3.2 STFL-syntax

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

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

```
var      ::= Identifier
number   ::= Number
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 in any compilation or interpretation process.

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

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

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

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

```
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 ()
```

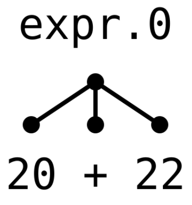


Figure 1.1: Parsetree of `20 + 22`

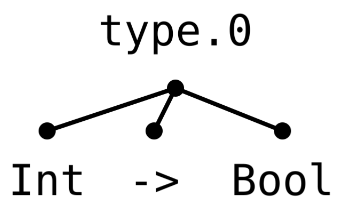


Figure 1.2: Parsetree of a function type

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 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.

Metafunctions have the following form:

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

The obligatory **signature** gives the name (`f`), followed by what syntactic forms the input parsetrees should have. The last element of the type[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.
- Second, they assign variables, which can be used to construct a new parsetree.
- Third, they can deconstruct large parsetrees, allowing finegrained pattern matching withing the parsetrees branches.
- Fourth, advanced searching behaviour is implemented in the form of evaluation contexts.

[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`.

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

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

Expr	As pattern
`x`	Captures the argument as the name. If multiple are used in the same pattern, the captured arguments should be the same or the match fails.
`_`	Captures the argument and ignores it
`42`	Argument should be exactly this number
`"Token"`	Argument should be exactly this string
`func(arg0, arg1, ...)`	Evaluates the function, matches if the argument equals the result. Can only use variables which are declared left of this pattern
`!func:type(arg0, ...)`	Evaluates the builtin function, matches if the argument equals the result. Can only use variables which are declared left of this pattern
`(expr or pattern:type)`	Check that the argument is an element of `type`
`e[expr or pattern]`	Matches the parsetree with `e`, searches a subpart in it matching `pattern`
`a "b" (nested)`	Splits the parse tree in the appropriate parts, pattern matches the subparts

Expr	Name	As expression
`x`	Variable	Recalls the parsetree associated with this variable
`_`	Wildcard	*Not defined*
`42`	Number	This number
`"Token"`	Literal	This string
`func(arg0, arg1, ...)`	Function call	Evaluate this function
`!func:type(arg0, ...)`	Builtin function call	Evaluate this builtin function, let it return a `type`
`(expr or pattern:type)`	Ascription	Checks that an expression is of a type. Bit useless 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 parse tree

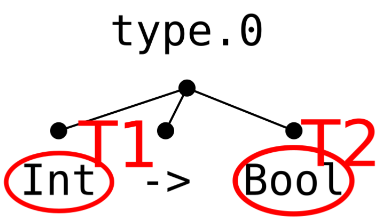


Figure 1.3: Pattern matching of `Int -> Bool`

1.4.1 Domain and codomain

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

```
 Functions
===========


domain                            : type -> type
domain("(" T ")")                 = domain(T)
domain(("(" T1 ")") "->" T2)      = T1
domain(T1 "->" T2)                = T1

codomain                          : type -> type
codomain("(" T ")")               = codomain(T)
codomain(T1"->" ("(" T2 ")"))     = T2
codomain(T1 "->" T2)              = T2
```

Recall the parsetree generated by parsing `Int -> Bool` against `type`. If this parsetree were used as input into the `domain` function, it would fail to match the first pattern (as the parsetree does not contain parentheses) nor would it match the second pattern (again are parentheses needed). The third pattern matches, by assigning `T1` and `T2`, as can seen in figure 1.3. `T1` is extracted and returned by `domain`, which is, by definition the domain of the `type`.

Analogously, the more advanced parsetree representing `Int -> Bool -> Bool` will be deconstructed the same way, as visible in figure 1.4, again capturing the domain withing `T1`.

The deconstructing behaviour of patterns can be observed when `(Int -> Bool) -> Bool` is used as argument for `domain`. It matches the second clause, deconstructing the part left of the arrow (`(Int -> Bool)`) and matching it against the embedded pattern `"(" T1 ")"`, as visualised in figure 1.5, capturing the domain *without parentheses* in `T1`.

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

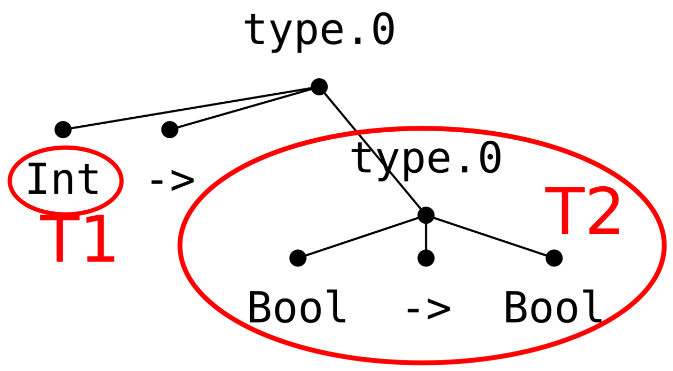


Figure 1.4: Pattern matching of `Int -> Bool -> Bool`

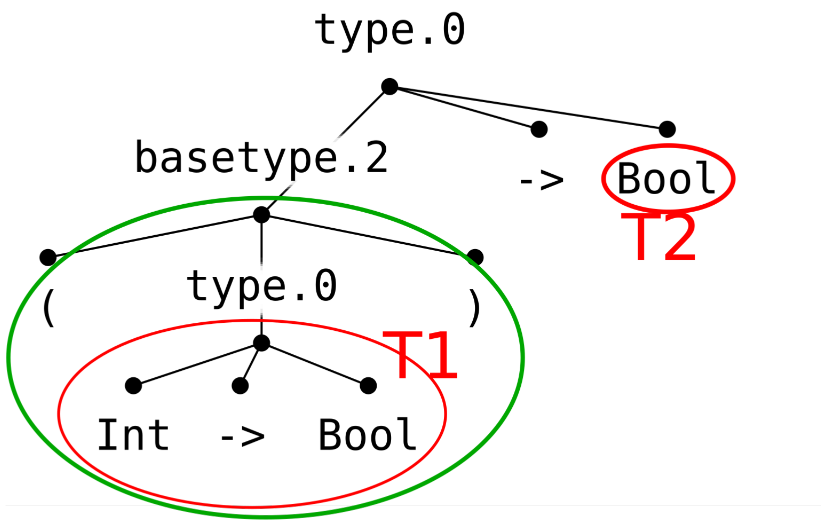


Figure 1.5: Pattern matching of `(Int -> Bool) -> Bool` . Matching in the subpattern is denoted with green

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, we present how operational semantics can be implemented and offer operational semantics 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 or equivalents; or merely by the calculation of the end result of an expression, where the effect is created by the human interpreting the result.

The meaning of any program 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

The first way to let a target program 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. While such translation is necessary to create programs which run as fast as possible on real hardware, it complicates matters for theoretical purposes. Using this approach, proving properties about the target language would involve first modeling the host language, proving an analogous property of the host language, followed by proving that the translation preserves the property.

The second way to give meaning to a program, is 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 these 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 tries to give meaning to the entire target program by giving meaning to each of the parts. This can be done in a inherently syntactic way, thus without leaving ALGT. Expressions and functional languages (such as `5 + 6`) can be evaluated by replacing the parsetree with the result (`11`), while imperative programs can handled by creating a syntactic for representing the state of the computer (which might contain a variable store, the output stream, ...).

While all of the above semantical approaches are possible within ALGT, structural operational semantics (or operational semantics for short) is the most practical one. The main ingredient -the basic parts of the language- are already there in the form of the syntax - only the transformation of the parsetree should be denoted.

All these semantics, especially structural operational semantics, can be constructed using natural deduction rules. Natural deduction rules allow us to describe relations in a straightforward and structured way. This can be seen in the next part, where this technique is applied to give STFL meaning.

1.5.2 Declaring smallstep

A transformation is in essence a relation between the current and the next state. In order to evaluate STFL-expressions, we will construct a relation which rewrites expressions as `1 + 1` into `2`, giving the end result. This relation is called $\rightarrow$, with signature `expr` $\times$`expr`. Examples of elements in this relation are `1 + 1` $\rightarrow$`2`, `If True Then 0 Else 1` $\rightarrow$`0`, `...`

Defining this relation 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 the $\rightarrow$ relation is as following:

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

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, a symbol is given for the relation, between parentheses: `(→)`. Then, the types of the arguments are given, by `: expr (in), expr (out)`, denoting that $\rightarrow$ is a relation in `expr` $\times$`expr`. Each argument also has a mode, one of `in` or `out`, written between parentheses. This is to help the tool proving the relation: given `1 + 1`, the relation can easily deduce that this is rewritten to `2`. However, given `2`, 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. Relations with no output arguments are allowed, an example of this would be a predicate checking for equality.

The last part, `Pronounced as "small step"` is documentation. It serves as human readable name, hinting the role of the relation within the language for users of the programming which are not familiar with commonly used symbols. While this is optional, it is strongly recommended to write.

1.5.3 Natural deduction rule

The actual implementation of *smallstep* is given by multiple natural deduction rules, where a **natural deduction rule** describes one facet of complex relations.

A natural deduction rule is declared in the following form:

```
 pred1  pred2   pred3 ...
-------------------------------- [Rule0Name]
 (relation) arg0 , arg1
```

The most important part of a rule is written below the line, which states that states that `(arg0, arg1)` is in `relation`. `arg0` and `arg1` can be advanced expressions as seen with functions. On an input argument, the expression will be used as pattern match and will construct a variable store. Based on this variable store, the output arguments can be constructed (if output arguments exists).

However, `(arg0, arg1)` will only be part of `relation` if all the predicates (`pred1`, `pred2`, ...) are valid. The predicates, written above the line and seperated by tab characters, act as extra guards. They have the same form as a rule conclusion:`(predRel) predArg0 predArg1 ...`. ALGT will attempt to proof each predicate, by constructing the input arguments for `predRel` and searching for an applicable rule.

As `(predRel) argIn argOut` might have output arguments as well, pattern matching on output of `predRel` might introduce new variables, which can be used in further predicates or in the end conclusion of the rule.

Predicates might also be of the form `x:syntacticForm` or `x = y`. `x:syntacticForm` checks wethers `x` is a parsetree formed with `syntacticForm`. `x = y` passes when `x` is identical to `y`.

At last, relations with two arguments can also be written infix: `arg0 relation arg1` is equivalent to `(relation) arg0, arg1`.

1.5.4 Defining Smallstep

With the tools presented above, implementing the `smallstep` relation is feasible.

If-then-else and parentheses

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

```

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

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

Analogously, the case for `False` is implemented:

```

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

Another straightforward rule is the removal of parentheses:

```

----------------                               [EvalParens]
 "(" e ")" → e
```

Plus

Plus 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`.

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

Type ascription

Type ascription is the syntactic form checking that a an expression is of a certain type. If this is the case, evaluation continues as if this ascription were nonexistent. If not, 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`.

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

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

```
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. Substitution can be done with the builtin function `!subs`.

The argument should have the correct 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`.

```
arg:value       arg :: T
------------------------------------------------- [EvalLamApp]
("(" "\\" var ":" T "." e ")") arg → !subs:e(var, arg, e)
```

Evaluating contexts

At last, the given evaluation rules can't handle nested expressions, such as `1 + (2 + 3)`. There is no rule telling ALGT that it should first evaluate this expression to `1 + 5`, after which it can calculate to `6`.

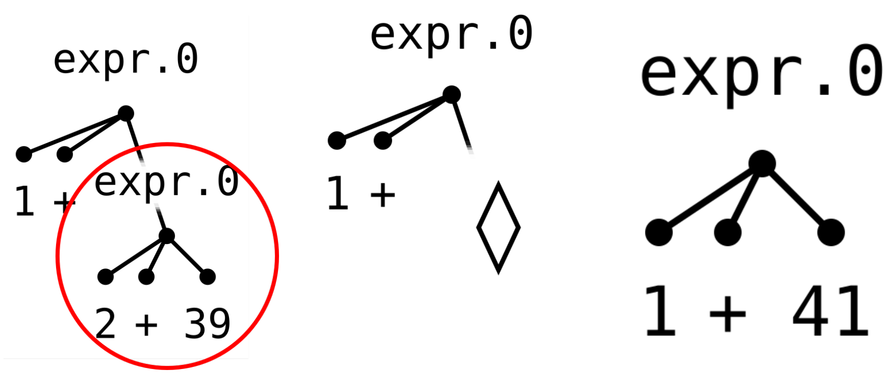
It is rather cumbersome to introduce rules for each positon where a syntactic form might be reduced. For `+`, this would need two extra rules (`e0 "+" e →e1 "+" e` and `e "+" e0 →e "+" e1`), `If`-expressions would need an additional three rules, ... This clearly does not scale.

In order to scale this, a convergence rule is added:

```
e0 → e1
-----------------                         [EvalCtx]
e[e0] → e[e1]
```

This rule uses a *evaluation context*: `e[e0]`. This pattern will capture the entire parsetree 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 1.6.



(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 the subexpression removed

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

Figure 1.6: 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


-----------------------------------------        [EvalIfFalse]
 "If" "False" "Then" e1 "Else" e2 → e2


 arg:value      arg :: T
------------------------------------------------ [EvalLamApp]
 ("(" "\\" var ":" T "." e ")") arg → !subs:e(var, arg, e)
```

The relation → needs a single input argument, so we might run it 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 expression 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
```

1.5.5 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 part, a typechecker for STFL is constructed, using the same notation as introduced in the previous part. Just like the relation →, which related two expressions, a relation :: is defined which relates expression to its type. The relation :: is thus a relation in `expr` ×`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:

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

A new relation `(⊢)` is introduced takes, apart from the expression, a typing environment as well 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*

1.5.6 Declaration of :: and ⊢

Just like defining the semantics, we begin by declaring the relations.

Both `::` and ⊢ have an `expr` and a `type` argument. In both cases, `expr` can't be of mode `out`, as an infinite number of expressions exist of 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 is has mode `out` 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:

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

1.5.7 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 `{}`:

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

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:

```
 b:bool
-----------------            [Tbool]
 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:

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

Typing parens

When an expression of the form `(e)` should be typed, the type of the whole is the same as the type of the enclosed expression. This can be expressed as a rule tying these elements together:

```
 Γ ⊢ e, T
-------------------          [TParens]
 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 type of the expression 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 three predicates of the rule:

```
 Γ ⊢ c, "Bool"  Γ ⊢ e1, Tl      Γ ⊢ e2, Tr      Tl == Tr
----------------------------------------------------------        [TIf]
 Γ ⊢ "If" c "Then" e1 "Else" e2, Tl
```

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 expressions.

Lambda abstractions

The type of a lambda expression is the function type, with domain the type of the argument and codomain 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:

```
 ((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, we introduced a helper function earlier which calculates exactly this. At last, we check that the argument is of the expected type, namely the domain of the function.

This results in the following rule:

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

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, Tl      Γ ⊢ e2, Tr      Tl == Tr
--------------------------------------------------------        [TIf]
```

```
 Γ ⊢ "If" c "Then" e1 "Else" e2, T1



 ((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)
```

With these rules, the expressions which were evaluated in the previous chapter, can be typed. The proofs leading to their typing, are given below:

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


            2 : number  3 : number  T1 = Int = T2 T1 = Int = T2
            ----------- ----------- ------------- -------------
1 : number  {} ⊢ 2, Int {} ⊢ 3, Int Int == Int    Int == Int    T1 =
----------- --------------------------------------------------- ----
{} ⊢ 1, Int {} ⊢ 2 + 3, Int                                     Int
Int == Int
--------------------------------------------------------------------
{} ⊢ 1 + 2 + 3, Int
--------------------------------------------------------------------
1 + 2 + 3 :: Int
```

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


                      1 : number            T1 = Int = T2 T1 = Int =
--------------------- --------------------- ------------- ----------
x : Int , {} ⊢ x, Int x : Int , {} ⊢ 1, Int Int == Int    Int == Int
--------------------------------------------------------------------
x : Int , {} ⊢ x + 1, Int
41 : number  T1 = Int = T2
--------------------------------------------------------------------
{} ⊢ ( \ x : Int . x + 1 ), Int -> Int
{} ⊢ 41, Int Int == Int
--------------------------------------------------------------------
{} ⊢ ( \ x : Int . x + 1 ) 41, Int
--------------------------------------------------------------------
( \ x : Int . x + 1 ) 41 :: Int
```