Contents

0.1 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.

0.1.1 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.

0.1.2 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 →, with signature `expr` ×`expr`. Examples of elements in this relation are `1 + 1` →`2`, `If True Then 0 Else 1` →`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 → 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 → is a relation in `expr` ×`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.

0.1.3 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`.

0.1.4 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 `Numbers`; 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`.

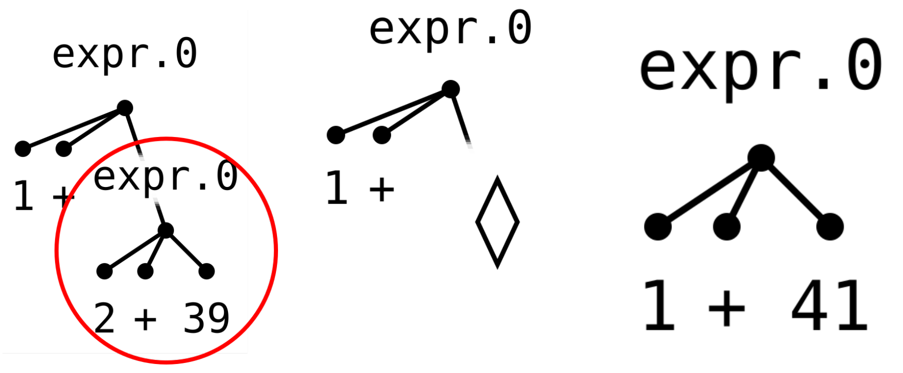
It is rather cumbersome to introduce rules for each positon where a syntactic form might be reduced. For `+`, this would need two extra rules (`e0 "+" e →e1 "+" e` and `e "+" e0 →e "+" e1`), `If`-expressions would need an additional three rules, . . . This clearly does not scale.

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.



(a) Parsetree of which `1 + (2 + 39)`, 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: An evaluation context where a subtree is replaced by its corresponding evluated expression

Semantics

The semantics of the STFL language can be captured in 7 straightforward natural deduction rules, in a straightforward and human readable format.

```
 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
```

0.1.5 Typechecker

0.2 Automatic checks