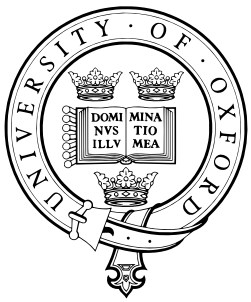
Computing Science Group

A Functional Implementation of the Formal Template Language

Nicolas Wu

CS-RR-09-10



Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD

A Functional Implementation of the Formal Template Language

Nicolas Wu

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
nicolas.wu@comlab.ox.ac.uk

Abstract. There has been growing interest in using the Z notation to describe design patterns and to encourage model driven development, but these are often expressed in terms of instances, rather than in a more general form. Instead of relying on the interpretation of instances, the Formal Template Language (FTL) has been used with Z as a means of capturing patterns in a framework that generates code on instantiation, thereby allowing reuse at the level of modelling and verification in a formal way. Until now, the instantiation of these templates has been manual. We present an implementation of the FTL in Haskell that allows the automatic generation of sentences from templates and evaluation environments. Our implementation uses Haskell and Happy (a functional parser generator for Haskell) to generate a parser that performs semantic analysis on given templates within specific environments to produce instantiations. By construction our implementation is faithful to the FTL specification in Z, by exploiting the commonalities between this specification and Haskell itself.

1 Introduction

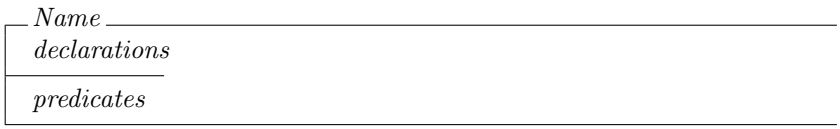
In recent years there has been a growing interest within the software engineering community in using abstractions and frameworks that assist in the correct design of systems. In particular, it is now widely accepted that describing often occurring problems and their solutions in terms of patterns [1] is both fruitful and beneficial, and this methodology is becoming part of the common software engineering toolbox. Formal methods are also gaining popularity, and specification languages such as the Z notation [2] are increasingly being considered as a useful way of outlining a solution during the design stage.

It is of little surprise, then, that there has been interest in using patterns within formal methods [3], and indeed a catalogue of Z patterns has been compiled [4, 5], thus encouraging the reuse of commonly found abstractions within that notation. Typically, a pattern is described in terms of an instance, where the particular instance serves as an example that best captures the form of the pattern. Describing patterns in this way makes them easily understood, and leaves the details of an application explicit, but the lack of formality with regards to

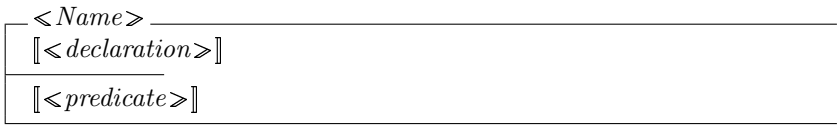
the general form of the pattern makes it impossible to automate the generation of instances from these descriptions. At present, semi-formal notations such as UML [6] are used in this context, where its formal syntax combined with its pragmatic approach to design has helped it to become universally accepted. However, UML is not without its flaws, and its lack of formal semantics makes it difficult to reason about systems with the precision of a fully formal notation like Z.

Recent work by Amálio, Stepney and Polack has focused on developing a fully formal framework for representing patterns. They have introduced the Formal Template Language, which allows the representation of patterns of formal development and enabling reasoning at the pattern level using meta-proof [7]. The FTL is able to capture the form of sentences of arbitrary languages, and has been used with Z as a means of generalising schemas in a way that is more flexible and expressive than the use of Z generics. There have been several successful applications of the FTL: it has been used to describe an object oriented method of formal design based on patterns [8]; in combination with UML and Z to assist with model-driven development [9]; and more recently we have been usingit as a means of describing relational database meta-models in Z.

At the heart of the FTL is the desire to formally delineate aspects of a language from parts that are in fact part of the meta-language. For example, a textbook introduction to Z might demonstrate the syntax of schemas using the following pseudocode:



However, the identifiers *Name*, *declarations*, and *predicates* are meant to act as placeholders for other text, and should not form part of the schema verbatim. The FTL disambiguates this by using the following notation:



Here placeholders are specifically enclosed by ≪ ≫ brackets, and lists of placeholders are enclosed by ⟦ ⟧ brackets. The concept behind the FTL is not remarkably complex, yet this level of precision in specification becomes particularly useful when trying to formally describe patterns that occur within a language or notation, where a simple augmentation of existing syntax is welcomed.

The applications of the FTL have yet to reach their full potential, since they are all somewhat hindered by the lack of tool support. Indeed, the significance of a generative framework like the FTL lies in its ability to capture often used patterns of development, and remove the burden of instantiating these patterns from the developer. Yet in the absence of automated template instantiation, the

developer is left with the error prone, and somewhat tedious task of manually instantiating templates.

In this report, we present an implementation of the FTL in a functional style using Haskell [10] to define semantics, and Happy [11] to generate a parser. The generated parser returns a data structure that represents a template, and this can be evaluated in the context of an environment to generate an appropriate instance. Both the syntax and the semantics of the FTL have been defined by Amálio using a combination of BNF notation for the syntax, and Z for the semantics, and we show that our approach lends itself well to implementing code that has been specified using those notations.

Our implementation is useful since it allows the mechanised generation of sources that can then be analysed by the tool of the target language. This is particularly useful for formal languages like Z, where type checking and automated proofs can be performed by other tools once the templates have been instantiated.

2 The Formal Template Language

Before we begin describing our implementation, we provide a brief introduction to the syntax of the FTL in BNF and Z, as introduced by Amálio [12]. We describe the various elements of syntax in turn, along with the data representations used for its particular elements. In this exposition BNF is used to describe the syntax of the FTL, and Z to describe the environment data structure.

2.1 Syntax

The FTL is made up of four different constructs, *text*, *placeholders*, *lists*, and *choices*. All of the identifiers in the FTL are made up of symbols drawn from the target language, $SYMB$, or from a set of identifiers I. In Z, we write these two sets as follows, where we also provide the definition of Str:

$$[SYMB, I] \qquad\qquad Str == \operatorname{seq} SYMB$$

We have defined Str as a convenience, since sequences of symbols are often used in the FTL specification.

Text The most basic element of the FTL is a fragment of text, which can be considered to be a sequence of characters from the target language. We denote a text element by T, and it can be defined as:

$$T ::= Str$$

As a basic unit, these text fragments are useful only when used in conjunction with the other template constructs.

Placeholder A placeholder is used to hold a template variable, which is instantiated by text denoted in its associated *instantiation environment*. Each placeholder contains an identifier I that is made distinct from the surrounding text by using angled brackets as shown below. The identifier I is instantiated by an environment Env which is effectively a mapping between I and Str:

$$\ll I \gg \qquad\qquad Env == I \pfun Str$$

When the semantic evaluator encounters a placeholder, it is replaced by the string that its identifier maps to within the environment in context. For example, consider the following template, alongside an instantiation mapping (shown as a set of maplets):

$$\ll x \gg : \ll t \gg \qquad\qquad \{x \mapsto a, t \mapsto \mathbb{N}\}$$

This trivially produces the text $a : \mathbb{N}$ as its instantiation.

List What makes the FTL particularly useful is the use of lists of placeholders. A list term LT is enclosed in template list brackets:

$$L ::= [\![LT]\!]$$

The list terms within lists are actually sequences of atoms, where an atom A is either text, a placeholder, or another list. A list is evaluated within the context of a tree structure of environments:

$$\begin{aligned} LT &::= A \mid A\ LT \qquad\qquad TreeEnv ::= tree\langle\!\langle Env \times \operatorname{seq} TreeEnv \rangle\!\rangle \\ A &::= \ll I \gg \mid T \mid L \end{aligned}$$

The *TreeEnv* structure is required since its enclosing *Env* provides the mappings for placeholders within the current scope, and the sequence of *TreeEnv* provides mappings for lists within that scope.

This is more easily explained with an example (adapted from [12]), where we consider the following template, alongside a final instantiation:

$$\begin{array}{ll} [\ll s \gg] & [X] \\ [\![\ll x \gg ::= \ll v \gg [\![\mid \ll v \gg]\!]]\!] & A ::= a_1 \mid a_2 \mid a_3 \\ & B ::= b_1 \mid b_2 \end{array}$$

The text on the right is the result of applying the following environment to the template:

$$\begin{array}{l} tree(\{s \mapsto X\}, \\ \quad \langle tree(\{x \mapsto A, v \mapsto a_1\}, \\ \quad\quad \langle tree(\{v \mapsto a_2, \langle\rangle), tree(\{v \mapsto a_3\}, \langle\rangle)\rangle), \\ \quad tree(\{x \mapsto B, v \mapsto b_1\rangle, \\ \quad\quad \langle tree(\{v \mapsto b_2\}, \langle\rangle)\rangle)\rangle) \end{array}$$

By using this tree of environments, arbitrarily nested list instantiations like the one above are made possible.

Lists also support a notation that allows us to define a separator SEP, and an empty instantiation to the list EI, which are fragments of text that are to be placed between list elements, and when the content of the list is empty. We use Λ to denote the empty string, giving us the following definition for a list L, and how the two list constructors relate:

$$L ::= [\![LT]\!]_{(SEP,EI)} \mid [\![LT]\!] \qquad\qquad [\![LT]\!] = [\![LT]\!]_{(\Lambda,\Lambda)}$$

We cover the details of the semantics of lists in more detail in Section 3.

Choice The final construct that the FTL supports is the choice of instantiation. Indeed, any FTL expression E is a sequence made up of atoms and choices:

$$E ::= A \mid C \mid A\ E \mid C\ E$$

A choice C can denote either an optional element or multiple choice, and is expressed as follows, where a choice list CL is the choice between at least two elements:

$$C ::= (\!|E|\!)^? \mid (\!|CL|\!) \qquad\qquad GEnv == \operatorname{seq}\mathbb{N} \times TreeEnv$$
$$CL ::= E \,[\!]\, E \mid E \,[\!]\, CL$$

In order to instantiate a choice of environments, we must further extend the definition of an instantiation environment, and use $GEnv$ for this purpose. The environment $GEnv$ uses a sequence of integers, one for each choice environment in turn, that dictates which choice is taken.

For optional choice environments, we use 0 to indicate that the choice is not taken, and 1 to indicate otherwise. For example, consider the following template:

$$(\!|\ll x \gg : \ll t \gg|\!)^?$$

This could be instantiated with either of the following environments:

$$(\langle 0\rangle, tree(\emptyset, \langle\rangle)) \qquad\qquad (\langle 1\rangle, tree(\{x \mapsto a, t \mapsto \mathbb{N}\}, \langle\rangle))$$

The first environment produces no text, and the second is the result of ordinary substitution, which yields $a : \mathbb{N}$. Multiple choice environments are instantiated similarly, where the desired choice is indicated by the appropriate number indicated in the environment.

In the next section we describe our implementation of the formal language using Haskell. This implementation is equivalent to the specification given by Amálio that we have discussed here, and shown in full in Appendix A. Our initial implementation closely resembles the specification in its structure and we later refine the data structure for a more efficient version.

2.2 Parser

We have seen how the syntax of the FTL can be described in terms of BNF production rules, and this gives an overview of how the terminal tokens relate to one another, and how structure is imposed by the grammar. The specification also comes with a description in Z, which indicates appropriate functional constructors for the various structural aspects of the grammar. These are used when defining the semantics of the FTL. The description of the FTL syntax is shown alongside the Z constructors in Figure 1; a correspondence that we can make use of in our functional definitions.

$$
\begin{array}{rll}
E ::= A \mid C \mid A\ E \mid C\ E & & eat\langle\langle A \rangle\rangle \mid ech\langle\langle C \rangle\rangle \mid eats\langle\langle A \times E \rangle\rangle \mid echs\langle\langle C \times E \rangle\rangle \\
C ::= (\!| E |\!)^? \mid (\!| CL |\!) & & och\langle\langle E \rangle\rangle \mid mch\langle\langle CL \rangle\rangle \\
CL ::= E\ [\!]\ E \mid E\ [\!]\ CL & & chs\langle\langle E \times E \rangle\rangle \mid lchs\langle\langle E \times CL \rangle\rangle \\
A ::= \ll I \gg \mid T \mid L & & param\langle\langle I \rangle\rangle \mid tx\langle\langle Str \rangle\rangle \mid ls\langle\langle L \rangle\rangle \\
L ::= [\![LT]\!]_{(SEP,EI)} \mid [\![LT]\!] & & list\langle\langle LT \times Str \times Str \rangle\rangle \mid listr\langle\langle LT \rangle\rangle \\
LT ::= A \mid A\ LT & & at\langle\langle A \rangle\rangle \mid lat\langle\langle A \times LT \rangle\rangle \\
 & & \vdash \forall\ lt : LT \bullet listr\ lt = list(lt, \langle\rangle, \langle\rangle)
\end{array}
$$

Fig. 1. The formal syntax of the FTL, with BNF notation on the left, and Z specification on the right.

We can translate the Z description of constructors directly to a series of Haskell datatypes, by ensuring each constructor begins with an upper case letter, and Currying datatypes that are made up of Cartesian products by removing parenthesis and allowing functions of higher order type.[1]

$$
\begin{array}{ll}
\textbf{data}\ E & = Eat\ A \mid Ech\ C \mid Eats\ A\ E \mid Echs\ C\ E \\
\textbf{data}\ C & = Och\ E \mid Mch\ CL \\
\textbf{data}\ CL & = Chs\ E\ E \mid Lchs\ E\ CL \\
\textbf{data}\ A & = Param\ I \mid Tx\ String \mid Ls\ L \\
\textbf{data}\ L & = List\ LT\ String\ String \\
\textbf{data}\ LT & = At\ A \mid Lat\ A\ LT \\
\textbf{type}\ I & = String
\end{array}
$$

These datatypes allow us to describe how the parser should link parsed tokens to data constructions in Happy, shown in Figure 2. The correspondence between the mixed BNF and Z notation and the Happy grammar specification makes the translation from specification to implementation remarkably simple.

A Happy grammar consists of various terminal and non-terminal symbol recursions, each with its associated production code, denoted by a pair of braces

[1] This Currying is not essential, but results in code that is of preferable style.

```
E  : A                                          { Eat $ 1 }
   | C                                          { Ech $ 1 }
   | A E                                        { Eats $ 1 $ 2 }
   | C E                                        { Echs $ 1 $ 2 }
C  : '(|' E '|)?'                               { Och $ 2 }
   | '(|' CL '|)'                               { Mch $ 2 }
CL : E '[]' E                                   { Chs $ 1 $ 3 }
   | E '[]' CL                                  { Lchs $ 1 $ 3 }
A  : '<|' String '|>'                           { Param (toI $ 2) }
   | String                                     { Tx $ 1 }
   | L                                          { Ls $ 1 }
L  : '[|' LT '|]_' '(' String ',' String ')'    { List $ 2 $ 5 $ 7 }
   | '[|' LT '|]'                               { List $ 2 [] [] }
LT : A                                          { At $ 1 }
   | A LT                                       { Lat $ 1 $ 2 }
```

Fig. 2. The FTL grammar described in Happy.

and written to the right of the appropriate token match rule. When a pattern is matched by a rule, the appropriate production is created, based on what is in between the braces. The production code is ordinary Haskell, where matches in the syntax are referred to by position using $\$n$ as its notation, where n denotes the syntactic element to match.

Our production code is nothing more than the appropriate type constructor required for a particular rule. The only exception is when we want an identifier within the *Param* constructor, where we use the *toI* function to make sure that the identifier is valid.

```
toI :: String → I
toI s = takeWhile (¬ ∘ isSpace) $ dropWhile isSpace s
```

This function removes any leading whitespace, and consumes following characters until the next occurrence of whitespace, or the input string is empty. The consumed characters are used as the identifier.

We stray from the specification grammar at only one point, since the definition of L should also include a constructor *Listr LT*, but we do not include it since we encode the equivalence

```
Listr lt = List lt [] []
```

directly into our parser. When the syntax for a *Listr* is detected by the parser, the syntax tree is populated with the equivalent *List* type instead.

Using the grammar in Figure 2, Happy produces a function that we have named *parser* that takes a list of tokens and returns the result of the first rule that is defined: in our case, the rule named E, which always returns a data constructor for the type E that we introduced earlier.

2.3 Lexer

The Happy grammar in Figure 2 makes use of various tokens to define the terminal symbols. This is done in another section of the Happy grammar file, shown in Figure 3. Each terminal symbol is associated to a token that is typically generated by a lexer, and these are listed side by side.

```
%token
  String {TString$$}
  '<|'   {TBSubs}    '|>' {TESubs}    '[]'  {TChoice}
  '(|'   {TBChoice}  '|)' {TEChoice}  '|)?' {TEChoice2}
  '[|'   {TBList}    '|]' {TEList}    '|]_' {TEList2}
  '('    {TBParen}   ')'  {TEParen}   ','   {TComma}
```

Fig. 3. Terminal symbol definitions in Happy, each alongside its corresponding token.

It is the job of the lexer to transform a particular input string, or sequence of symbols, into appropriate tokens. For completeness, we define a lexer that looks for a sequence of symbols that match an ASCII representation of the various FTL constructs, and treats all other characters as template text.

```
lexer :: String → [Token]
lexer = compact ∘ lexer′
lexer′                          :: String → [Token]
lexer′ []                       = []
lexer′ ('<' : '|' : cs)         = TBSubs : lexer′ cs
lexer′ ('|' : '>' : cs)         = TESubs : lexer′ cs
lexer′ ('(' : '|' : cs)         = TBChoice : lexer′ cs
lexer′ ('|' : ')' : cs)         = TEChoice : lexer′ cs
lexer′ ('|' : ')' : '?' : cs)   = TEChoice2 : lexer′ cs
lexer′ ('[' : '|' : cs)         = TBList : lexer′ cs
lexer′ ('|' : ']' : '_' : cs)   = TEList2 : param cs
lexer′ ('|' : ']' : cs)         = TEList : lexer′ cs
lexer′ ('[' : ']' : cs)         = TChoice : lexer′ cs
lexer′ (c : cs)                 = TChar c : lexer′ cs
```

The lexer itself depends on a function *compact* that removes lists of tokens of type $TChar$, and replaces them with a single token of type $TString$.

```
compact                                 :: [Token] → [Token]
compact []                              = []
compact (TChar c : [])                  = [TString [c]]
compact (TChar c : ts)                  = compact (TString [c] : ts)
compact (TString s : TChar c : ts)      = compact (TString (c : s) : ts)
```

$$
\begin{array}{ll}
compact\ (TString\ s : ts) & = TString\ (reverse\ s) : compact\ ts \\
compact\ (t : ts) & = t : compact\ ts
\end{array}
$$

This allows us to avoid the explicit transformation of sequences of characters to strings within the FTL grammar definition. This definition accumulates sequences of characters in a leading $TString$ structure. Appending characters to the head of a list takes constant time, but reverses the sequence order. The original sequence is recovered by reversing the list when no more characters are encountered.

The function $param$ is used to extract two parameters after a list type. This is a recursive definition whose final continuation is the function $lexer'$.

$$
\begin{array}{ll}
param & :: String \rightarrow [Token] \\
param\ (\texttt{'\textbackslash\textbackslash'} : \texttt{'\textbackslash\textbackslash'} : cs) & = TChar\ \texttt{'\textbackslash\textbackslash'} : param\ cs \\
param\ (\texttt{'\textbackslash\textbackslash'} : \texttt{'\{'} : cs) & = TChar\ \texttt{'\{'} : param\ cs \\
param\ (\texttt{'\textbackslash\textbackslash'} : \texttt{'\}'} : cs) & = TChar\ \texttt{'\}'} : param\ cs \\
param\ (\texttt{'\{'} : cs) & = TBParen : param\ cs \\
param\ (\texttt{'\}'} : \texttt{'\{'} : cs) & = TComma : param\ cs \\
param\ (\texttt{'\}'} : cs) & = TEParen : lexer'\ cs
\end{array}
$$

We have defined our lexer this way for convenience; a more complete solution would also encode a mechanism for escaping the various reserved sequences of characters that have been used to represent the FTL constructs. Moreover, the input sequence need not be our stylised ASCII representation, and a more typical method of inputting specifications might be through a LaTeX script. Since the FTL specification makes no reference to the exact nature of the lexicographic analysis, we make no effort to continue its development here, although a more complex lexer could easily be generated either manually, or by using a tool like Alex [13] that generates Haskell tokenisers based on regular expressions.

With a lexer and parser fully defined, we have effectively created a means of transforming sequences of characters into Haskell datatypes, representing the structure of an FTL template, that can then be evaluated by a semantic analyser.

3 Semantic analyser

With our grammar in place, and a means of taking structured text into an instantiation of that grammar, we now focus on giving appropriate semantics to the structure. In this section, we use the Z specification for the FTL found in Appendix A to guide our development.

3.1 Environment

Our first task is to define the appropriate data structure that will hold the environments with which the templates will be evaluated. Again, the translation between the Z specification and Haskell is very natural, with the code below corresponding to the Z specification of Amálio [12] that we discussed in Section 2.

$$
\begin{array}{ll}
\textbf{type}\ Env & = Map\ I\ String \\
\textbf{data}\ TreeEnv & = Tree\ Env\ [TreeEnv] \\
\textbf{data}\ GEnv & = GEnv\ [Integer]\ TreeEnv
\end{array}
$$

An alternative definition of Env would make it a synonym for $I \rightarrow String$, using it as a function directly. However, we have chosen to use a Map instead, since this provides a convenient way of creating new mappings from lists of associations, and also has an efficient implementation. This allows for simple instantiation of a new Env, and also gives us a reasonable mechanism for handling incomplete mappings using $Maybe$ constructs.

3.2 Variable Extraction

The semantic analyser needs to be able to ascertain whether or not a particular variable is in context for a particular expression. The functions $\mathcal{V}_{\mathcal{A}}$, $\mathcal{V}_{\mathcal{LT}}$, $\mathcal{V}_{\mathcal{C}}$, $\mathcal{V}_{\mathcal{CL}}$, and $\mathcal{V}_{\mathcal{E}}$ reflect those in the Z specification in a functional style.[2]

$$
\begin{array}{ll}
\mathcal{V}_{\mathcal{A}} & :: A \rightarrow [I] \\
\mathcal{V}_{\mathcal{A}}\ (Param\ i) & = [i] \\
\mathcal{V}_{\mathcal{A}}\ (Tx\ s) & = [\,] \\
\mathcal{V}_{\mathcal{A}}\ (Ls\ l) & = [\,] \\
\mathcal{V}_{\mathcal{LT}} & :: LT \rightarrow [I] \\
\mathcal{V}_{\mathcal{LT}}\ (At\ a) & = \mathcal{V}_{\mathcal{A}}\ a \\
\mathcal{V}_{\mathcal{LT}}\ (Lat\ a\ lt) & = union\ (\mathcal{V}_{\mathcal{A}}\ a)\ (\mathcal{V}_{\mathcal{LT}}\ lt) \\
\mathcal{V}_{\mathcal{C}} & :: C \rightarrow [I] \\
\mathcal{V}_{\mathcal{C}}\ (Och\ e) & = \mathcal{V}_{\mathcal{E}}\ e \\
\mathcal{V}_{\mathcal{C}}\ (Mch\ cl) & = \mathcal{V}_{\mathcal{CL}}\ cl \\
\mathcal{V}_{\mathcal{CL}} & :: CL \rightarrow [I] \\
\mathcal{V}_{\mathcal{CL}}\ (Chs\ e_1\ e_2) & = union\ (\mathcal{V}_{\mathcal{E}}\ e_1)\ (\mathcal{V}_{\mathcal{E}}\ e_2) \\
\mathcal{V}_{\mathcal{CL}}\ (Lchs\ e\ cl) & = union\ (\mathcal{V}_{\mathcal{E}}\ e)\ (\mathcal{V}_{\mathcal{CL}}\ cl) \\
\mathcal{V}_{\mathcal{E}} & :: E \rightarrow [I] \\
\mathcal{V}_{\mathcal{E}}\ (Eat\ a) & = \mathcal{V}_{\mathcal{A}}\ a \\
\mathcal{V}_{\mathcal{E}}\ (Ech\ c) & = \mathcal{V}_{\mathcal{C}}\ c \\
\mathcal{V}_{\mathcal{E}}\ (Eats\ a\ e) & = union\ (\mathcal{V}_{\mathcal{A}}\ a)\ (\mathcal{V}_{\mathcal{E}}\ e) \\
\mathcal{V}_{\mathcal{E}}\ (Echs\ c\ e) & = union\ (\mathcal{V}_{\mathcal{C}}\ c)\ (\mathcal{V}_{\mathcal{E}}\ e)
\end{array}
$$

These functional definitions follow on very naturally from the Z specification. The most noticeable change is that we use lists, rather than sets of I. This is a convenience that later affords us brevity when checking membership of the list. Since lists preserve multiplicity of elements, we need to ensure that the our list maintains the uniqueness of its elements. This property is enforced by the function *union* which is the only function used to compose lists.

[2] The function names here and throughout the document have been formatted for exposition, and are easily replaced by valid Haskell identifiers.

3.3 Semantic Evaluation

Our final task is to provide the functions that generate strings, given an appropriate template and environment. Again, the translation from specification to implementation is with little effort.

Since we decided to implement the type Env with a Map, we dereference the value of env with value i by using $env\,!\,i$, rather than function application.

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{A}} & & :: A \to TreeEnv \to String \\
\mathcal{M}_{\mathcal{A}}\ (Tx\ t) & (Tree\ env\ lte) & = t \\
\mathcal{M}_{\mathcal{A}}\ (Param\ i) & (Tree\ env\ lte) & = env\,!\,i \\
\mathcal{M}_{\mathcal{A}}\ (Ls\ l) & (Tree\ env\ lte) & = \mathcal{M}_{\mathcal{L}}\ l\ lte
\end{array}
$$

The largest changes between the specification found in the appendix and the implementation are in the function that follows. Here, we have refactored the singleton and non-empty cases of $\mathcal{M}_{\mathcal{L}}$ into a single non-empty case. This refactoring is made valid since

$$\mathcal{M}_{\mathcal{L}}\ (List\ (Lat\ (Tx\ sep)\ lt)\ [\,]\ [\,])\ [\,] \equiv [\,]$$

and so the singleton and non-empty case agree, and can reduce to the non-empty case alone.

Possibly the most significant change is the refinement of the specification:

$$\neg \mathcal{V}_{\mathcal{LT}}\ lt \cap \mathrm{dom}\ env = \emptyset$$

to its implementation as:

$$any\ (flip\ member\ env)\ (\mathcal{V}_{\mathcal{LT}}\ lt)$$

Here we have made the most of our decision to use $[I]$ rather than $Set\ I$ as the return value of $\mathcal{V}_{\mathcal{LT}}$, since the function any maps a predicate onto a list, and returns $true$ if any result of the application of the predicate yields $true$.

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{L}} & & :: L \to [TreeEnv] \to String \\
\mathcal{M}_{\mathcal{L}}\ (List\ lt\ sep\ ei)\ [\,] & & = ei \\
\mathcal{M}_{\mathcal{L}}\ (List\ lt\ sep\ ei)\ ((Tree\ env\ lte) : ts) & & \\
\quad \mid any\ (flip\ member\ env)\ (\mathcal{V}_{\mathcal{LT}}\ lt) & & = \mathcal{M}_{\mathcal{LT}}\ lt\ (Tree\ env\ lte) +\!\!+ \\
 & & \quad \mathcal{M}_{\mathcal{L}}\ (List\ (Lat\ (Tx\ sep)\ lt)\ [\,]\ [\,])\ ts \\
\quad \mid otherwise & & = ei
\end{array}
$$

In the functions that follow the changes in translation are all trivial.

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{LT}} & & :: LT \to TreeEnv \to String \\
\mathcal{M}_{\mathcal{LT}}\ (At\ a) & t & = \mathcal{M}_{\mathcal{A}}\ a\ t \\
\mathcal{M}_{\mathcal{LT}}\ (Lat\ a\ lt) & t & = \mathcal{M}_{\mathcal{A}}\ a\ t +\!\!+ \mathcal{M}_{\mathcal{LT}}\ lt\ t \\
\mathcal{M}_{\mathcal{CL}} & & :: CL \to Integer \to E \\
\mathcal{M}_{\mathcal{CL}}\ (Chs\ e_1\ e_2)\ 1 & & = e_1
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{CL}}\ (Chs\ e_1\ e_2)\ 2 & = & e_2 \\
\mathcal{M}_{\mathcal{CL}}\ (Lchs\ e\ cl)\ 1 & = & e \\
\mathcal{M}_{\mathcal{CL}}\ (Lchs\ e\ cl)\ n & = & \mathcal{M}_{\mathcal{CL}}\ cl\ (n-1) \\
\mathcal{M}_{\mathcal{C}} & :: & C \to Integer \to E \\
\mathcal{M}_{\mathcal{C}}\ (Och\ e)\ 0 & = & Eat\ (Tx\ [\,]) \\
\mathcal{M}_{\mathcal{C}}\ (Och\ e)\ n & = & e \\
\mathcal{M}_{\mathcal{C}}\ (Mch\ cl)\ n & = & \mathcal{M}_{\mathcal{CL}}\ cl\ n
\end{array}
$$

As before, we have refactored the following definition to remove redundant singleton list clauses, which turn out to be equivalent to non-empty cases.

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{E}} & :: & E \to GEnv \to String \\
\mathcal{M}_{\mathcal{E}}\ (Eat\ a)\ (GEnv\ ns\ t) & = & \mathcal{M}_{\mathcal{A}}\ a\ t \\
\mathcal{M}_{\mathcal{E}}\ (Ech\ c)\ (GEnv\ (n:ns)\ t) & = & \mathcal{M}_{\mathcal{E}}\ (\mathcal{M}_{\mathcal{C}}\ c\ n)\ (GEnv\ ns\ t) \\
\mathcal{M}_{\mathcal{E}}\ (Eats\ a\ e)\ (GEnv\ ns\ t) & = & \mathcal{M}_{\mathcal{A}}\ a\ t +\!\!+ \mathcal{M}_{\mathcal{E}}\ e\ (GEnv\ ns\ t) \\
\mathcal{M}_{\mathcal{E}}\ (Echs\ c\ e)\ (GEnv\ (n:ns)\ t) & = & \mathcal{M}_{\mathcal{E}}\ (\mathcal{M}_{\mathcal{C}}\ c\ n +\!\!+_E e)\ (GEnv\ ns\ t)
\end{array}
$$

$$
\begin{array}{lll}
(+\!\!+_E) & :: & E \to E \to E \\
(Eat\ a)\ +\!\!+_E\ e & = & Eats\ a\ e \\
(Ech\ c)\ +\!\!+_E\ e & = & Echs\ c\ e \\
(Eats\ a\ e_1) +\!\!+_E\ e_2 & = & Eats\ a\ (e_1 +\!\!+_E e_2) \\
(Echs\ c\ e_1) +\!\!+_E\ e_2 & = & Echs\ c\ (e_1 +\!\!+_E e_2)
\end{array}
$$

The changes we have made throughout have been somewhat superficial, and for the most part were motivated by a desire for elegance, rather than by necessity. The ease of development of a functional implementation from a mixed BNF and Z specification speaks for itself, and we have highlighted how using Happy and Haskell have made this remarkably simple.

4 Example

At this point we demonstrate the evaluation of an expression in an environment using the $\mathcal{M}_{\mathcal{E}}$ function as appropriate. For the purpose of this example, we use the following template, which is made up of most of the elements of the FTL:

$$(\!|\![\!\ll x \gg : \ll y \gg;\ ]\!]\ [\!]\ [\![\ll x \gg : \ll y \gg \to \ll z \gg;\ ]\!]|\!)$$

Instantiating this template demonstrates the use of choice, lists, placeholders, and text. In our discussion above, we implemented the FTL tokens for these elements using an ASCII representation, so we use this representation of the above template in the Haskell code that follows.[3] The value of t is the result of first tokenising the input string using our *lexer* function, and then using

[3] This rendering is rather difficult to read, and is the consequence of our simplified lexer. The production version of this tool uses LaTeX input and is therefore easier to read, so for the purposes of this example we concede this eyesore.

the function *parser* generated by Happy, to return a structure of type E that represents the template.

```
t = (parser ∘ lexer)
  "(|[|<|x|> : <|y|>; |][][|<|x|> : <|y|> -> <|z|>; |]|)"
```

We then define the tree environment that we use to evaluate the template:

```
env =
  Tree empty [Tree (fromList [("x","a"),("y","A"),("z","B")]) [],
              Tree (fromList [("x","b"),("y","C"),("z","D")]) []]
```

Here the function $fromList$ takes a list of pairs, and returns a map where the first element of a pair becomes a key, with the value found in the second element of the pair.

Since the template includes choice, we demonstrate the two alternatives:

```
M_E t (GEnv [1] env) = "a : A; b : C; "
M_E t (GEnv [2] env) = "a : A -> B; b : C -> D; "
```

This example does little to provide full validation of our tool, but serves as a brief demonstration of its capabilities. In fact, using Haskell as the implementation language has gained us a hidden benefit: the text above showing the result of applying $\mathcal{M}_{\mathcal{E}}$ was generated entirely automatically using `lhs2TeX`, a tool that is able to execute fragments of Haskell within a source script before generating LaTeX code. Since the source of this very report is a Haskell script that contains all the definitions discussed so far, we not only have the assurance that the definitions are well typed (since it has been checked by a Haskell compiler), but this document has the precise result of our template instantiation rendered automatically.

In the following section we consider how the implementation might be improved and optimised further.

5 Refactoring and Optimisation

5.1 Left Recursion is Happier

Our main optimisation is to do with the way that Happy generates compilers. The parsers produced as a result of a left recursion are more efficient than those that are a result of a right recursion. A left recursion results in a constant stack parser, whereas a right recursion results in a parser that requires space equivalent to the length of the list parsed. Since the grammar of the FTL is relatively simple, making all rules left-recursive is not a difficult task, and results in what is shown in Figure 4, where recursions found in E, CL, and LT have all been modified.

Unfortunately, this refactoring adds an inconvenient complication, since new elements that are on the right hand side of the grammar are appended to the left

$$
\begin{aligned}
E' &::= A' \mid C' \mid E'\ A' \mid E'\ C' \\
C' &::= (\!|E'|\!)^{?} \mid (\!|CL'|\!) \\
CL' &::= E' \ [\!]\ E' \mid CL' \ [\!]\ E' \\
A' &::= \ll I \gg \mid T \mid L' \\
L' &::= [\![LT']\!]_{(SEP,EI)} \mid [\![LT']\!] \\
LT' &::= A' \mid LT'\ A'
\end{aligned}
$$

Fig. 4. A left-factored version of the FTL BNF syntax.

of a constructed datatype. This results in the reversal of the element order, and the resulting datatypes must be reversed again to regain the original sequence. This can be achieved by defining a reversal function for the types that have had their rules altered. For example, the reverse function for the type E is the following[4]:

$$
\begin{aligned}
&reverse_E :: E \to E \\
&reverse_E\ (Eat\ a) = Eat\ a \\
&reverse_E\ (Ech\ c) = Ech\ c \\
&reverse_E\ (Eats\ a\ e) = reverse_E\ e +\!\!+_E\ Eat\ a \\
&reverse_E\ (Echs\ c\ e) = reverse_E\ e +\!\!+_E\ Ech\ c
\end{aligned}
$$

Similar definitions for CL and LT are also required, which in turn need implementations of specialised concatenation. To complete this refactoring, we rewrite the Happy grammar to produce what is shown in Figure 5.

Alternatively, it would be possible to redefine the basic datatypes with constructor parameters reversed, but this would have required a reimplementation of the various $\mathcal{V}$ and $\mathcal{M}$ functions to reflect this change, which would be considerably removed from the original specification.

Since our changes are entirely in the grammar production rules, and the return types are unmodified, our semantic evaluator $\mathcal{M}_{\mathcal{E}}$ does not need to change to function correctly with this optimised parser.

5.2 Native Datatypes

The initial FTL specification in Z [12] suggested a concrete syntax for the various data structures, using free type definitions to directly represent each of the BNF constructs. The flexibility of Haskell datatype constructors allowed us to use these free type definitions almost verbatim, and this in turn allowed a very natural translation between the Z semantics and their implementation. Another approach would be to represent the syntactic elements using native Haskell datatypes.

[4] This is an inefficient version of reverse that does not use an accumulator. We use it here for its simple definition.

```
E' : A'                                          { Eat $1 }
   | C'                                          { Ech $1 }
   | E' A'                                       { Eats $1 $2 }
   | E' C'                                       { Echs $1 $2 }
C' : '(|' E' '|)?'                               { Och (reverse_E $2) }
   | '(|' CL' '|)'                               { Mch (reverse_CL $2) }
CL': E' '[]' E'                                  { Chs (reverse_E $3) (reverse_E $1) }
   | CL' '[]' E'                                 { Lchs $1 (reverse_E $3) }
A' : '<|' String '|>'                            { Param (toI $2) }
   | String                                      { Tx $1 }
   | L'                                          { Ls $1 }
L' : '[|' LT' '|]_' '(' String ',' String ')'    { List (reverse_LT $2) $5 $7 }
   | '[|' LT' '|]'                               { List (reverse_LT $2) [] [] }
LT': A'                                          { At $1 }
   | LT' A'                                      { Lat $2 $1 }
```

Fig. 5. The FTL' grammar described in Happy.

For example, the type LT expressed in Figure 1 could be more naturally defined as follows, using an extended BNF notation, where a + indicates that there is one or more of the preceding element.

$$LT ::= A+$$

Indeed, this definition is found in an early version of the FTL [8]. Viewing the definition this way makes it rather obvious that LT ought to be implemented using a non-empty list of type A, and so we might suggest this as a more logical representation:

$$\textbf{data}\ LT' = [A']$$

Similarly, the type E can be expressed using the following syntax:

$$E ::= (A \mid C)+$$

And this indicates that E ought to be implemented as a list of elements that can be either A or C:

$$\textbf{type}\ E' = [Either\ A'\ C']$$

Continuing in this way, an alternative set of datatypes that represents the FTL syntax could be used, where synonyms of the familiar native datatypes are favoured.[5] Arguably, such definitions would have been the natural starting point for our program.

[5] Strictly speaking, we have introduced some generalisations here that were not originally specified — the type E' now contains an empty list, whereas the type of E does not permit this. We would therefore have to take care to handle this case appropriately in the redefinition of both $\mathcal{V}_\mathcal{E}$ and $\mathcal{M}_\mathcal{E}$, and likewise for other definitions that introduce lists.

Defining the basic types in this style complicates the translation from specification to implementation, but results in an implementation that can exploit the rich library of functions that are provided by Haskell. However, there is little incentive to do so in this case, since we scarcely make use of the library functions.

6 Conclusion

We have demonstrated the implementation of the FTL in a functional style using Haskell and Happy, and as far as we are aware, this is the first working implementation of the FTL. In our implementation we have been careful to adhere to the specification set out by Amálio, and have shown that the translation from Z and BNF to Haskell and Happy is rather natural.

Other parser generators like Antlr and Yacc were also considered for use, but the perspicuous nature of Haskell is appealing for its strength in exposition and for its clarity. In addition, the implementation of a specification in Z lends itself quite naturally to a functional style of programming. Other functional approaches such as OCaml and its parser generator Ocamlyacc were also considered, and Haskell was favoured for its lighter syntactic requirements.

On a practical note, using this tool has proved to be useful for database templates, but we find that the means of inputting instantiation environments using raw data structures is somewhat cumbersome. This is not a problem with our implementation *per se*, since such structures are part of the design of the FTL. Further theoretical work concerning the FTL itself would be required, with the aim of facilitating the creation of instantiation environments in a notation that more closely resembles the target language, and that has fewer syntactic requirements. To solve this, we plan to extend the use of templates to consider not only how to output structured text, but as a means of inputting text too.

Our implementation has also brought to our attention other aspects of the FTL that could be improved. For example, the choice of templates currently has two constructors; one to create optional choices, and the other to create multiple choices. It is not clear why these two constructors have not been unified, since one could easily imagine a single choice constructor with multiple choices that could be optionally ignored using the same mechanism that optional choices currently take. As such, this simplification was not implemented since our aim was to design a tool that was fully compliant with the original FTL. To this end, our implementation has shown itself to be entirely successful.

Acknowlegements

The author is grateful to Andrew Simpson for his comments on an earlier draft of this report.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley (1995)
2. Spivey, J.M.: The Z notation: A Reference Manual. Prentice-Hall (1992)
3. Stepney, S., Polack, F., Toyn, I.: An outline pattern language for Z: five illustrations and two tables. In Bert, D., Bowen, J.P., King, S., Walden, M., eds.: ZB2003: Third International Conference of B and Z Users, Turku, Finland. Volume 2651 of LNCS., Springer (2003) 2–19
4. Stepney, S., Polack, F., Toyn, I.: A Z patterns catalogue I: specification and refactorings, v0.1. Technical Report YCS-2003-349, Department of Computer Science, University of York (January 2003)
5. Valentine, S.H., Stepney, S., Toyn, I.: A Z patterns catalogue II: definitions and laws, v0.1. Technical Report YCS-2004-383, Department of Computer Science, University of York (October 2004)
6. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Higher Education (2004)
7. Amálio, N., Stepney, S., Polack, F.: A Formal Template Language Enabling Metaproof. In Misra, J., Nipkow, T., Sekerinski, E., eds.: Proceedings of FM 2006. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 252–267
8. Amálio, N., Stepney, S., Polack, F.: Modular UML semantics: Interpretations in Z based on templates and generics. In Van, H.D., Liu, Z., eds.: FACS'03 Workshop on Formal Aspects of Component Software, Pisa, Italy, September 2003. Volume 284 of UNU/IIST Technical Report. (2003)
9. Amálio, N., Polack, F., Stepney, S.: Frameworks Based on Templates for Rigorous Model-driven Development. Electr. Notes Theor. Comput. Sci. **191** (2007) 3–23
10. Jones, S.L.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
11. Gill, A., Marlow, S.: Happy: The parser generator for Haskell (June 2009) `http://www.haskell.org/happy/`.
12. Amálio, N.: Generative Frameworks for Rigorous Model-Driven Development. PhD thesis, Department of Computer Science, Univ of York (2006)
13. Marlow, S.: Alex: A lexical analyser generator for Haskell (June 2009) `http://www.haskell.org/alex/`.

A FTL Semantics in Z

This appendix shows the various definitions of the FTL semantic functions, as given by Amálio [12].

A.1 Auxiliary Definitions

$[I, SYMB]$
$Str == \text{seq}\, SYMB$

$$
\begin{array}{|l}
\_ \mathbin{+\!\!+}_E \_ : E \times E \to E \\
\hline
\forall\, a : A;\ e : E \bullet (eat\ a) \mathbin{+\!\!+}_E\ e = eats\,(a, e) \\
\forall\, c : C;\ e : E \bullet (ech\ c) \mathbin{+\!\!+}_E\ e = echs\,(c, e) \\
\forall\, a : A;\ e_1, e_2 : E \bullet eats\,(a, e_1) \mathbin{+\!\!+}_E\ e_2 = eats\,(a, e_1 \mathbin{+\!\!+}_E\ e_2) \\
\forall\, c : C;\ e_1, e_2 : E \bullet echs\,(c, e_1) \mathbin{+\!\!+}_E\ e_2 = echs\,(c, e_1 \mathbin{+\!\!+}_E\ e_2)
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{V}_{\mathcal{A}} : A \to \mathbb{P}\, I \\
\hline
\forall\, i : I \bullet \mathcal{V}_{\mathcal{A}}\,(param\ i) = \{i\} \\
\forall\, s : Str \bullet \mathcal{V}_{\mathcal{A}}\,(tx\ s) = \emptyset \\
\forall\, l : L \bullet \mathcal{V}_{\mathcal{A}}\,(ls\ l) = \emptyset
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{V}_{\mathcal{LT}} : LT \to \mathbb{P}\, I \\
\hline
\forall\, a : A \bullet \mathcal{V}_{\mathcal{LT}}\,(at\ a) = \mathcal{V}_{\mathcal{A}}\ a \\
\forall\, a : A;\ le : LT \bullet \mathcal{V}_{\mathcal{LT}}\,(lat\,(a, le)) = \mathcal{V}_{\mathcal{A}}\ a \cup \mathcal{V}_{\mathcal{LT}}\ le
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{V}_{\mathcal{C}} : C \to \mathbb{P}\, I \\
\mathcal{V}_{\mathcal{CL}} : CL \to \mathbb{P}\, I \\
\mathcal{V}_{\mathcal{E}} : E \to \mathbb{P}\, I \\
\hline
\forall\, e : E \bullet \mathcal{V}_{\mathcal{C}}\,(och\ e) = \mathcal{V}_{\mathcal{E}}\ e \\
\forall\, cl : CL \bullet \mathcal{V}_{\mathcal{C}}\,(mch\ cl) = \mathcal{V}_{\mathcal{CL}}\ cl \\
\forall\, e_1, e_2 : E \bullet \mathcal{V}_{\mathcal{CL}}\,(chs\,(e_1, e_2)) = \mathcal{V}_{\mathcal{E}}\ e_1 \cup \mathcal{V}_{\mathcal{E}}\ e_2 \\
\forall\, e : E;\ cl : CL \bullet \mathcal{V}_{\mathcal{CL}}\,(lchs\,(e, cl)) = \mathcal{V}_{\mathcal{E}}\ e \cup \mathcal{V}_{\mathcal{CL}}\ cl \\
\forall\, a : A \bullet \mathcal{V}_{\mathcal{E}}\,(eat\ a) = \mathcal{V}_{\mathcal{A}}\ a \\
\forall\, c : C \bullet \mathcal{V}_{\mathcal{E}}\,(ech\ c) = \mathcal{V}_{\mathcal{C}}\ c \\
\forall\, a : A;\ e : E \bullet \mathcal{V}_{\mathcal{E}}\,(eats\,(a, e)) = \mathcal{V}_{\mathcal{A}}\ a \cup \mathcal{V}_{\mathcal{E}}\ e \\
\forall\, c : C;\ e : E \bullet \mathcal{V}_{\mathcal{E}}\,(echs\,(c, e)) = \mathcal{V}_{\mathcal{C}}\ c \cup \mathcal{V}_{\mathcal{E}}\ e
\end{array}
$$

A.2 Semantic Functions

$$
\begin{array}{l}
Env == I \nrightarrow Str \\
TreeEnv ::= tree\langle\!\langle Env \times \operatorname{seq} TreeEnv \rangle\!\rangle \\
GEnv == \operatorname{seq} \mathbb{N} \times TreeEnv
\end{array}
$$

$$
\begin{array}{l}
\mathcal{M}_{\mathcal{A}} : A \to \mathit{TreeEnv} \nrightarrow \mathit{Str} \\
\mathcal{M}_{\mathcal{L}} : L \to \mathrm{seq}\, \mathit{TreeEnv} \nrightarrow \mathit{Str} \\
\mathcal{M}_{\mathcal{LT}} : \mathit{LT} \to \mathit{TreeEnv} \nrightarrow \mathit{Str} \\
\hline
\forall\, t : \mathit{Str};\ \mathit{env} : \mathit{Env};\ \mathit{lte} : \mathrm{seq}\, \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{A}}\, (\mathit{tx}\ t)\, (\mathit{tree}\, (\mathit{env}, \mathit{lte})) = t \\
\forall\, i : I;\ \mathit{env} : \mathit{Env};\ \mathit{lte} : \mathrm{seq}\, \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{A}}\, (\mathit{param}\ i)\, (\mathit{tree}\, (\mathit{env}, \mathit{lte})) = \mathit{env}\ i \\
\forall\, l : L;\ \mathit{env} : \mathit{Env};\ \mathit{lte} : \mathrm{seq}\, \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{A}}\, (\mathit{ls}\ l)\, (\mathit{tree}\, (\mathit{env}, \mathit{lte})) = \mathcal{M}_{\mathcal{L}}\ l\ \mathit{lte} \\
\forall\, \mathit{le} : \mathit{LT};\ \mathit{ld}, \mathit{let} : \mathit{Str} \bullet \mathcal{M}_{\mathcal{L}}\, (\mathit{list}\, (\mathit{le}, \mathit{ld}, \mathit{let}))\, \langle\rangle = \mathit{let} \\
\forall\, \mathit{le} : \mathit{LT};\ \mathit{ld}, \mathit{let} : \mathit{Str};\ \mathit{env} : \mathit{Env};\ \mathit{lte} : \mathrm{seq}\, \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{L}}\, (\mathit{list}\, (\mathit{le}, \mathit{ld}, \mathit{let}))\, \langle (\mathit{tree}\, (\mathit{env}, \mathit{lte}))\rangle = \\
\quad\quad \textbf{if}\ \neg \mathcal{V}_{\mathcal{LT}}\ \mathit{le} \cap \mathrm{dom}\ \mathit{env} = \emptyset \\
\quad\quad \textbf{then}\ \mathcal{M}_{\mathcal{LT}}\ \mathit{le}\, (\mathit{tree}\, (\mathit{env}, \mathit{lte})) \\
\quad\quad \textbf{else}\ \mathit{let} \\
\forall\, \mathit{le} : \mathit{LT};\ \mathit{ld}, \mathit{let} : \mathit{Str};\ \mathit{env} : \mathit{Env};\ \mathit{ltes}, \mathit{ltet} : \mathrm{seq}\, \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{L}}\, (\mathit{list}\, (\mathit{le}, \mathit{ld}, \mathit{let}))\, (\langle \mathit{tree}\, (\mathit{env}, \mathit{ltes})\rangle \frown \mathit{ltet}) = \\
\quad\quad \textbf{if}\ \neg \mathcal{V}_{\mathcal{LT}}\ \mathit{le} \cap \mathrm{dom}\ \mathit{env} = \emptyset \\
\quad\quad \textbf{then}\ \mathcal{M}_{\mathcal{LT}}\ \mathit{le}\, (\mathit{tree}\, (\mathit{env}, \mathit{ltes})) \frown \mathcal{M}_{\mathcal{L}}\, (\mathit{list}\, (\mathit{lat}\, ((\mathit{tx}\ \mathit{ld}), \mathit{le}, \langle\rangle, \langle\rangle))\ \mathit{ltet} \\
\quad\quad \textbf{else}\ \mathit{let} \\
\forall\, a : A;\ \mathit{te} : \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{LT}}\, (\mathit{at}\ a)\ \mathit{te} = \mathcal{M}_{\mathcal{A}}\ a\ \mathit{te} \\
\forall\, a : A;\ \mathit{le} : \mathit{LT};\ \mathit{te} : \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{LT}}\, (\mathit{lat}\, (a, \mathit{le}))\ \mathit{te} = \mathcal{M}_{\mathcal{A}}\ a\ \mathit{te} \frown \mathcal{M}_{\mathcal{LT}}\ \mathit{le}\ \mathit{te}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{M}_{\mathcal{CL}} : \mathit{CL} \to \mathbb{N}_1 \nrightarrow E \\
\hline
\forall\, e_1, e_2 : E \bullet \mathcal{M}_{\mathcal{CL}}\, (\mathit{chs}\, (e_1, e_2))\, 1 = e_1 \\
\forall\, e_1, e_2 : E \bullet \mathcal{M}_{\mathcal{CL}}\, (\mathit{chs}\, (e_1, e_2))\, 2 = e_2 \\
\forall\, e : E;\ \mathit{cl} : \mathit{CL};\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{CL}}\, (\mathit{lchs}\, (e, \mathit{cl}))\, n = \\
\quad \textbf{if}\ n = 1\ \textbf{then}\ e\ \textbf{else}\ \mathcal{M}_{\mathcal{CL}}\ \mathit{cl}\, (n - 1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{M}_{\mathcal{C}} : C \to \mathbb{N} \nrightarrow E \\
\hline
\forall\, e : E \bullet \mathcal{M}_{\mathcal{C}}\, (\mathit{och}\ e)\, 0 = \mathit{eat}\, (\mathit{tx}\, \langle\rangle) \\
\forall\, e : E;\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{C}}\, (\mathit{och}\ e)\, n = e \\
\forall\, \mathit{cl} : \mathit{CL};\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{C}}\, (\mathit{mch}\ \mathit{cl})\, n = \mathcal{M}_{\mathcal{CL}}\ \mathit{cl}\ n
\end{array}
$$

$$
\begin{array}{l}
\mathcal{M}_{\mathcal{E}} : E \to \mathit{GEnv} \to \mathit{Str} \\
\hline
\forall\, a : A;\ \mathit{chs} : \mathrm{seq}\, \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{E}}\, (\mathit{eat}\ a)\, (\mathit{chs}, \mathit{te}) = \mathcal{M}_{\mathcal{A}}\ a\ \mathit{te} \\
\forall\, c : C;\ n : \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \mathcal{M}_{\mathcal{E}}\, (\mathit{ech}\ c)\, (\langle n\rangle, \mathit{te}) = \mathcal{M}_{\mathcal{E}}\, (\mathcal{M}_{\mathcal{C}}\ c\ n)\, (\langle\rangle, \mathit{te}) \\
\forall\, c : C;\ n : \mathbb{N};\ \mathit{chs} : \mathrm{seq}\, \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{E}}\, (\mathit{ech}\ c)\, ((\langle n\rangle \frown \mathit{chs}), \mathit{te}) = \mathcal{M}_{\mathcal{E}}\, (\mathcal{M}_{\mathcal{C}}\ c\ n)\, (\mathit{chs}, \mathit{te}) \\
\forall\, a : A;\ e : E;\ \mathit{chs} : \mathrm{seq}\, \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{E}}\, (\mathit{eats}\, (a, e))\, (\mathit{chs}, \mathit{te}) = \mathcal{M}_{\mathcal{A}}\ a\ \mathit{te} \frown \mathcal{M}_{\mathcal{E}}\ e\, (\mathit{chs}, \mathit{te}) \\
\forall\, c : C;\ e : E;\ n : \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{E}}\, (\mathit{echs}\, (c, e))\, (\langle n\rangle, \mathit{te}) = \mathcal{M}_{\mathcal{E}}\, ((\mathcal{M}_{\mathcal{C}}\ c\ n) \mathbin{+\!\!+_E} e)\, (\langle\rangle, \mathit{te}) \\
\forall\, c : C;\ e : E;\ n : \mathbb{N};\ \mathit{chs} : \mathrm{seq}\, \mathbb{N};\ \mathit{te} : \mathit{TreeEnv} \bullet \\
\quad \mathcal{M}_{\mathcal{E}}\, (\mathit{echs}\, (c, e))\, ((\langle n\rangle \frown \mathit{chs}), \mathit{te}) = \mathcal{M}_{\mathcal{E}}\, ((\mathcal{M}_{\mathcal{C}}\ c\ n) \mathbin{+\!\!+_E} e)\, (\mathit{chs}, \mathit{te})
\end{array}
$$