1 Extended Language

In order to facilitate programming, it is useful to define an extended language. A program is first parsed into this language, and the resulting tree is translated into the regular abstract syntax.

This allows the core language to be concise, reducing the complexity of type inference and evaluation. Complex constructs (such as comprehensions and multi-parameter functions) can be included only in the extended language, and it suffices to provide a translation into the core language.

This translation does have the drawback of reducing the formality of evaluation. Since there are no evaluation rules for the additional constructs, it is impossible to prove the correctness of the translation rules. This does not in any way affect the correctness of the core language type inference and evaluation, and the advantages gained by this method far outweigh the drawbacks, so it is still a net positive to the language.

The following two sections will describe the abstract syntax tree for the extended language and how it translates to a syntax tree in the core language.

1.1 Abstract Syntax

The extended language has terms which are similar to (if not exactly the same as) ones existing in the core language. These terms are presented in their entirety here, and, since they are directly extracted from the core language, no explanation will be given for them.

$$
\begin{array}{lcll}
e' & ::= & func' & \\
 & | & e'_1 \; e'_2 & \\
 & | & x & \\
 & | & Builtin & \\
 & | & con & \\
 & | & \{l_1 : e'_1, \; \ldots \; l_n : e'_n\} & (n \geq 1) \\
 & | & \#l & \\
 & | & \#(e'_1, \; \ldots \; e'_n) & (n \geq 2) \\
 & | & raise & \\
 & | & \texttt{match}\; e' \;\texttt{with}\; match'_1, \ldots\; match'_n & (n \geq 1) \\
 & & & \\
match' & ::= & p' \to e' & \\
 & | & p' \;\texttt{when}\; e'_1 \to e'_2 &
\end{array}
$$

Most patterns are extracted from the core language without any differences, and are defined below.

$$\begin{array}{lcll}
p' & ::= & patt' & \\
 & \mid & patt' : T' & \\
 & & & \\
patt' & ::= & x & \\
 & \mid & \_ & \\
 & \mid & con\ p'_1,\ \ldots\ p'_n & \text{(constructor pattern, } n = \text{arity } con\text{)} \\
 & \mid & \{l_1 : p'_1,\ \ldots\ l_n : p'_n\} & (n \geq 1) \\
 & \mid & \{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\} & \text{(partial record, } n \geq 1\text{)}
\end{array}$$

Most types are, like patterns, extracted from the language.

$$\begin{array}{lcll}
T' & ::= & X^{Traits} & \\
 & \mid & conT\ T'_1,\ \ldots\ T'_n & (n = \text{arity } conT) \\
 & \mid & T'_1 \rightarrow T'_2 & \\
 & \mid & \{l_1 : T'_1,\ \ldots\ l_n : T'_n\} & (n \geq 1) \\
 & \mid & T'_1 \# T'_2 & Accessor
\end{array}$$

1.1.1 Additions

Type Aliases The first addition to the extended language is the concept of *type aliases*. These types are simple renames of existing types, and can be used in programs as a way to simplify type declarations.

$$\begin{array}{lcl}
T' & ::= & \cdots \\
 & \mid & \tau \\
 & & \\
\tau & ::= & \{\tau_0, \tau_1, \ldots\}
\end{array}$$

Conditional Expression A conditional expression, which translates to a `match` expression on the patterns *true* and *false*, has been added.

$$\begin{array}{lcl}
e' & ::= & \cdots \\
 & \mid & \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3
\end{array}$$

Multi-Parameter and Pattern Matching Functions Functions have been extended to allow multiple parameters, removing the necessity of declaring nested functions. These functions still require at least one parameter.

Furthermore, patterns are allowed as the definition of parameters.

$$\begin{array}{lcll}
func' & ::= & \cdots & \\
 & \mid & \texttt{fn}\ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & \mid & \texttt{rec}\ f : T\ \ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & \mid & \texttt{rec}\ f\ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1)
\end{array}$$

Declarations The `let` expression is also extended, and a new construction (*decl′*) is needed. Besides the basic value binding, 4 new function bindings are allowed. These correspond to all combinations of typed, untyped, recursive and non-recursive functions, with at least one parameter.

Along with value and function bindings, a new type alias binding was added. This binding creates a new type alias that can be used further down in the syntax tree.

$$\begin{array}{lcll}
e' & ::= & \cdots & \\
 & \mid & \texttt{let}\ decl'\ \texttt{in}\ e & \\
 & & & \\
decl' & ::= & p' = e' & \\
 & \mid & f\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & \mid & \texttt{rec}\ f\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & \mid & f : T'\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & \mid & \texttt{rec}\ f : T'\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & \mid & \texttt{type alias}\ \tau = T' &
\end{array}$$

Lists Although lists are supported by the base language with the :: (cons) and *nil* data constructors, they are not easy to use with only these terms.

The extended language provides a term to implicitly define a list, specifying all of its components between square brackets.

$$\begin{array}{lcll}
e' & ::= & \cdots & \\
 & \mid & [e'_1,\ \ldots\ e'_n] & (n \geq 0)
\end{array}$$

In a similar fashion, a specific pattern for lists is added.

$$\begin{array}{lcll}
patt' & ::= & \cdots & \\
 & \mid & [p'_1,\ \ldots\ p'_n] & (n \geq 0)
\end{array}$$

Range Using a similar construction to basic lists, *ranges* allow the programmer to specify a list of numbers without having to declare all of them explicitly.

There are two variations on ranges. The first is a simple range, providing the start and end values. This range creates a list with all integers starting from the first value, incrementing by one until the last value.

The second variation provides, along with the start and end values, the second value of the list. This allows the language to know what the increment of the range is. Besides allowing increments greater than 1, this also allows ranges that decrement from the start value until the end value.

$$\begin{array}{lcl}
e' & ::= & \cdots \\
 & \mid & [e'_1\ ..\ e'_2] \\
 & \mid & [e'_1,\ e'_2\ ..\ e'_3]
\end{array}$$

Comprehension V provides a very basic list comprehension syntax. This allows evaluating an expression for every value in an existing list, returning a list with the results of every evaluation.

$$\begin{array}{lll} e' & ::= & \cdots \\ & | & [e'_1 \text{ for } p' \text{ in } e'_2] \end{array}$$

Tuple Like lists, tuples are supported by the language through the Tuple n constructor. To allow easier creation of tuples, a new term is added to the extended language.

$$\begin{array}{lll} e' & ::= & \cdots \\ & | & (e'_1,\ \ldots\ e'_n) \qquad (n \geq 2) \end{array}$$

1.2 Translation

To actually evaluate or type check a program in the extended language, it must first be translated into the core language. This is done by a translation algorithm which, besides converting extended terms into core terms, also performs some additional safety checks.

A translation rule is of the form:

$$\gamma \vdash e' \Rightarrow e$$

where γ is the translation environment.

Besides translating expressions, the translation algorithm also translates types (T'), functions ($func'$), etc. All these translations will be described using the same format, and also use the same environment.

Enviroment Like evaluation and type inference, the translation algorithm requires an environment to properly function. This environment contains the following information:

1. Type aliases

   A mapping of type aliases to core types

2. Mapping of generated identifiers

   A mapping of identifiers to other identifiers. This is used because the translation algorithm can create new identifiers, and so it maps identifiers from the input expression to new identifiers in the output expression.

$$\begin{array}{lll} \gamma & ::= & (aliases, ids) \\ \\ aliases & ::= & \{\} \mid \{\tau \rightarrow T\} \cup aliases \qquad (n \in \mathbb{N}) \\ ids & ::= & \{\} \mid \{x_1 \rightarrow x_2\} \cup ids \end{array}$$

Below will be sections describing the translation algorithms for the different types of expressions in the extended language. As to avoid clutter, only the rules that perform some sort of computation or modification on the expression will be displayed. This means that rules such as:

$$\gamma \vdash \text{Int} \Rightarrow \text{Int} \tag{Tr-T-Int}$$

will not be provided.

Similarly, composite expressions that simply call the translation algorithm recursively on their sub-expressions, without any modification to structure, will be omitted. This includes rules such as:

$$\frac{\gamma \vdash T_1' \Rightarrow T_1 \qquad \gamma \vdash T_2' \Rightarrow T_2}{\gamma \vdash (T_1' \rightarrow T_2') \Rightarrow (T_1 \rightarrow T_2)} \tag{Tr-T-Func}$$

1.2.1 Type Translation

Given the fact that trivial translations are not provided, there is only one translation rule that governs type translations.

$$\frac{\gamma.aliases(\tau) = T}{\gamma \vdash \tau \Rightarrow T} \tag{Tr-T-Alias}$$

1.2.2 Pattern Translation

The algorithm for translating patterns works on lists of patterns instead of single patterns. This is done to allow verification of repeated identifiers in a list of patterns.

Since functions allow multiple patterns as parameters, a verification is done to ensure that there are no repeated identifiers in any of the parameters. Furthermore, composite patterns, such as a list or tuple pattern, also cannot repeat identifiers in their sub-patterns, as this would cause ambiguous binding.

This verification is done from left to right, composing a set of identifiers already used in the pattern. Each verification uses the set of identifiers from the previous verification, and so the complete set of used identifiers is created.

The translation also returns a modified translation environment. This environment has new identifier mappings, one for each x pattern found.

$$\frac{\begin{array}{c}\text{id}_0 = \emptyset \qquad \gamma_0 = \gamma \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p_i', \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\gamma \vdash [p_1',\ \dots\ p_n'] \Rightarrow [p_1,\ \dots\ p_n], \gamma_n} \tag{Tr-P}$$

To translate a single pattern, a list with only that pattern is created and then translated. This ensures that, even within a single pattern, no identifiers are allowed to repeat.

$$\frac{\gamma \vdash [p'] \Rightarrow [p], \gamma'}{\gamma \vdash p' \Rightarrow p, \gamma'} \quad \text{(Tr-P2)}$$

An auxiliary function, called "collectPatterns", is used in pattern translation. This function takes an extended pattern, a set of already used identifiers and a translation environment; and returns a core pattern and a new set of used identifiers.

This is the core of the pattern translation algorithm. If an identifier has already been used in a pattern (or list of patterns), the translation algorithm fails. If the identifier has not been used, a fresh identifier (guaranteed not to be in the environment) is generated. The original identifier is then associated to this new identifier and added to the environment.

$$\frac{x \notin \text{id} \qquad x' \textit{ is new}}{\text{collectPatterns}(x, \text{id}, \gamma) = x', \text{id} \cup \{x\}, \gamma \cup \{x \rightarrow x'\}} \quad \text{(Tr-P-X)}$$

$$\frac{x \notin \text{id} \qquad \gamma \vdash T' \Rightarrow T \qquad x' \textit{ is new}}{\text{collectPatterns}(x : T', \text{id}, \gamma) = x' : T, \text{id} \cup \{x\}, \gamma \cup \{x \rightarrow x'\}} \quad \text{(Tr-P-X2)}$$

Every other translation rule is a variation on iterating on sub-patterns to construct the list of used identifiers and final environment.

$$\text{collectPatterns}(\_, \text{id}, \gamma) = \_, \text{id}, \gamma \quad \text{(Tr-P-Ignore)}$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\text{collectPatterns}(\_ : T', \text{id}, \gamma) = \_ : T, \text{id}, \gamma} \quad \text{(Tr-P-Ignore2)}$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(con\ p'_1,\ \ldots\ p'_n, \text{id}, \gamma) = con\ p_1,\ \ldots\ p_n, \text{id}_n, \gamma_n} \quad \text{(Tr-P-Con)}$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \qquad \gamma \vdash T' \Rightarrow T \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(con\ p'_1,\ \ldots\ p'_n : T', \text{id}, \gamma) = con\ p_1,\ \ldots\ p_n : T, \text{id}_n, \gamma_n} \quad \text{(Tr-P-Con2)}$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\}, \text{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\}, \text{id}_n, \gamma_n}$$
(Tr-P-Record)

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \qquad \gamma \vdash T' \Rightarrow T \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\} : T', \text{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\} : T, \text{id}_n, \gamma_n}$$
(Tr-P-Record2)

$$\frac{\begin{array}{c}\text{id}_0 = \text{id}\\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(\{l_1 : p'_1,\ \dots\ l_n : p'_n, \dots\}, \text{id}, \gamma) = \{l_1 : p_1,\ \dots\ l_n : p_n, \dots\}, \text{id}_n\gamma_n} \quad (\textsc{Tr-P-PartRec})$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \qquad \gamma \vdash T' \Rightarrow T\\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\text{collectPatterns}(\{l_1 : p'_1,\ \dots\ l_n : p'_n, \dots\} : T', \text{id}, \gamma) = \{l_1 : p_1,\ \dots\ l_n : p_n, \dots\} : T, \text{id}_n, \gamma_n} \quad (\textsc{Tr-P-PartRec2})$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id}\\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\\ p = ::\ p_1\ (::\ p_2\ \dots(::\ p_n\ nil)\dots)\end{array}}{\text{collectPatterns}([p'_1,\ \dots\ p'_n], \text{id}, \gamma) = p, \text{id}_n, \gamma_n} \quad (\textsc{Tr-P-List})$$

$$\frac{\begin{array}{c}\text{id}_0 = \text{id} \qquad \gamma \vdash T' \Rightarrow T\\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p'_i, \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\\ p = ::\ p_1\ (::\ p_2\ \dots(::\ p_n\ nil)\dots)\end{array}}{\text{collectPatterns}([p'_1,\ \dots\ p'_n] : T', \text{id}, \gamma) = p : T, \text{id}_n, \gamma_n} \quad (\textsc{Tr-P-List2})$$

Typing Patterns Some expressions require that untyped patterns be transformed into typed patterns. For this, fresh variable types are generated.

$$\text{typeP}(patt' : T', \gamma) = patt' : T' \quad (\textsc{Typ-Patt})$$

$$\frac{X\ is\ new}{\text{typeP}(patt', \gamma) = patt' : X} \quad (\textsc{Typ-Patt2})$$

1.2.3 Function Translation

Since functions have undergone massive changes from the core language, they need complex translation rules.

$$\frac{\begin{array}{c}\gamma \vdash [p'_1,\ \dots\ p'_n] \Rightarrow [p_1,\ \dots\ p_n], \gamma'\\ \forall i \in [1,\ n]\ .\ x_i\ is\ new\\ \gamma' \vdash e' \Rightarrow e\\ m = \texttt{match}\ (x_1,\ \dots\ x_n)\ \texttt{with}\ (p_1,\ \dots p_n) \to e\\ f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{fn}\ p'_1,\ \dots\ p'_n \Rightarrow e') \Rightarrow f} \quad (\textsc{Tr-F-Fn})$$

$$\frac{\begin{array}{c}\gamma \vdash [p'_1,\ \dots\ p'_n] \Rightarrow [p_1,\ \dots\ p_n], \gamma'\\ f'\ is\ new \qquad \forall i \in [1,\ n]\ .\ x_i\ is\ new\\ \gamma' \cup \{f \to f'\} \vdash e' \Rightarrow e\\ m = \texttt{match}\ (x_1,\ \dots\ x_n)\ \texttt{with}\ (p_1,\ \dots p_n) \to e\\ fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f\ \ p'_1,\ \dots\ p'_n \Rightarrow e') \Rightarrow fn} \quad (\textsc{Tr-F-Rec})$$

$$\frac{\begin{array}{c}\forall i \in [1,n]\ .\ \text{typeP}(p'_i,\gamma) = \mathit{patt}'_i : T'_i \\ T'' = T'_2 \to T'_3 \to \cdots \to T'_n \\ \gamma \vdash [\mathit{patt}'_1 : T'_1,\ \ldots\ p'_n : T'_n] \Rightarrow [p_1,\ \ldots\ p_n], \gamma' \\ f'\ \mathit{is\ new} \qquad \forall i \in [1,\ n]\ .\ x_i\ \mathit{is\ new} \\ \gamma' \cup \{f \to f'\} \vdash e' \Rightarrow e \qquad \gamma' \vdash T'' \Rightarrow T \\ m = \texttt{match}\ (x_1,\ \ldots\ x_n)\ \texttt{with}\ (p_1,\ \ldots p_n) \to e \\ fn = \texttt{rec}\ f : T\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f : T'\ \ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow fn} \quad (\text{Tr-F-RecT})$$

When functions only have one parameter, the match expression does not use a tuple, since tuples must have at least 2 components.

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \qquad x\ \mathit{is\ new} \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ x\ \texttt{with}\ p \to e \\ f = \texttt{fn}\ x \Rightarrow m\end{array}}{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e') \Rightarrow f} \quad (\text{Tr-F-Fn1})$$

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \qquad x\ \mathit{is\ new} \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ x\ \texttt{with}\ p \to e \\ f = \texttt{rec}\ x \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ p' \Rightarrow e') \Rightarrow f} \quad (\text{Tr-F-Rec1})$$

$$\frac{\begin{array}{c}\text{typeP}(p',\gamma) = p'' \\ \gamma \vdash p'' \Rightarrow p, \gamma' \qquad x\ \mathit{is\ new} \\ \gamma' \vdash e' \Rightarrow e \qquad \gamma' \vdash T' \Rightarrow T \\ m = \texttt{match}\ x\ \texttt{with}\ p \to e \\ f = \texttt{rec}\ x : T \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ p' : T' \Rightarrow e') \Rightarrow f} \quad (\text{Tr-F-RecT1})$$

As a further efficiency improvement, that match expression is only created when the parameters of the function are patterns. If every parameter of the function is a regular identifier (without type information), then no match expression is necessary. This is only done if the function does not specify a return type.

$$\frac{\begin{array}{c}\gamma \vdash [x'_1,\ \ldots\ x'_n] \Rightarrow [x_1,\ \ldots\ x_n], \gamma' \\ \gamma' \vdash e' \Rightarrow e \\ f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e\end{array}}{\gamma \vdash (\texttt{fn}\ x'_1,\ \ldots\ x'_n \Rightarrow e') \Rightarrow f} \quad (\text{Tr-F-Fn2})$$

$$\frac{\begin{array}{c}\gamma \vdash [x'_1,\ \ldots\ x'_n] \Rightarrow [x_1,\ \ldots\ x_n], \gamma' \\ f'\ \mathit{is\ new} \qquad \gamma' \cup \{f \to f'\} \vdash e' \Rightarrow e \\ fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e\end{array}}{\gamma \vdash (\texttt{rec}\ f\ \ x'_1,\ \ldots\ x'_n \Rightarrow e') \Rightarrow fn} \quad (\text{Tr-F-Rec2})$$

1.2.4 Declaration Translation

The translation of declarations works differently from other translations. The result of translating a declaration is a set of associations between patterns and expressions; along with an updated translation environment.

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \\ \gamma \vdash e' \Rightarrow e\end{array}}{\gamma \vdash (p' = e') \Rightarrow \{p \rightarrow e\}, \gamma'} \quad (\text{Tr-Decl})$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\gamma \vdash \texttt{type alias}\ \tau = T' \Rightarrow \{\}, \gamma \cup \{\tau \rightarrow T\}} \quad (\text{Tr-Decl-Alias})$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{fn}\ p'_1, \ \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ \mathit{is\ new}\end{array}}{\gamma \vdash f\ p'_1, \ \ldots\ p'_n = e' \Rightarrow \{f' \rightarrow e\}, \gamma \cup \{f \rightarrow f'\}} \quad (\text{Tr-Decl-Func})$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{fn}\ p'_1, \ \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ \mathit{is\ new}\end{array}}{\gamma \vdash f : T'\ p'_1, \ \ldots\ p'_n = e' \Rightarrow \{f' \rightarrow e\}, \gamma \cup \{f \rightarrow f'\}} \quad (\text{Tr-Decl-Func2})$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{rec}\ f\ p'_1, \ \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ \mathit{is\ new}\end{array}}{\gamma \vdash \texttt{rec}\ f\ p'_1, \ \ldots\ p'_n = e' \Rightarrow \{f' \rightarrow e\}, \gamma \cup \{f \rightarrow f'\}} \quad (\text{Tr-Decl-Rec})$$

$$\frac{\begin{array}{c}\forall i \in [1,n]\ .\ \text{typeP}(p'_i, \gamma) = \mathit{patt}'_i : T'_i \\ T'' = T'_1 \rightarrow T'_2 \rightarrow \cdots \rightarrow T'_n \\ \gamma \vdash (\texttt{rec}\ f : T'\ \mathit{patt}'_1 : T'_1, \ \ldots\ \mathit{patt}'_n : T'_n \Rightarrow e') \Rightarrow e \\ f' \ \mathit{is\ new}\end{array}}{\gamma \vdash \texttt{rec}\ f : T'\ p'_1, \ \ldots\ p'_n = e' \Rightarrow \{f' : T'' \rightarrow e\}, \gamma \cup \{f \rightarrow f'\}} \quad (\text{Tr-Decl-Rec2})$$

1.2.5 Expression Translation

The conditional expression translates into a `match` expression, testing whether the first sub-expression (e_1) is *true* or *false* and evaluation e_2 or e_3, respectively.

$$\frac{\begin{array}{c}\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2 \qquad \gamma \vdash e'_3 \Rightarrow e_3 \\ e = \texttt{match}\ e_1\ \texttt{with}\ \mathit{true} \rightarrow e_2,\ \mathit{false} \rightarrow e_3\end{array}}{\gamma \vdash \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3 \Rightarrow e} \quad (\text{Tr-E-Cond})$$

The list expression translates into nested applications of the :: constructor, ending with the *nil* (empty list) constructor.

$$\frac{\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \qquad e = ::\ e_1\ (::\ e_2\ \ldots(::\ e_n\ nil)\ldots)}{\gamma \vdash [e'_1,\ \ldots\ e'_n] \Rightarrow e} \quad (\textsc{Tr-E-List})$$

Similarly, the tuple expression translates into a complete application of a tuple (Tuple n) constructor, where n is the number of elements in the expression.

$$\frac{\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \qquad e = (\ \ldots\ (\text{Tuple}\ n\ e_1)\ e_2)\ \ldots e_n)}{\gamma \vdash (e'_1,\ \ldots\ e'_n) \Rightarrow e} \quad (\textsc{Tr-E-Tuple})$$

When translating `let` expressions, the declaration is translated into an ordered set of associations between patterns and expressions. Nested `let` expressions are then created with these associations.

$$\frac{\gamma \vdash decl \Rightarrow \{p_1 \to e_1,\ \ldots\ p_n \to e_n\}, \gamma' \qquad \gamma' \vdash e' \Rightarrow e \qquad ret = (\texttt{let}\ p_1 = e_1\ \texttt{in} \cdots \texttt{let}\ p_n = e_n\ \texttt{in}\ e)}{\gamma \vdash \texttt{let}\ decl\ \texttt{in}\ e' \Rightarrow ret} \quad (\textsc{Tr-E-Let})$$

There are two variations of ranges: one with an implicit step and one with an explicit step. Both of these rely on the existing of the function "range", which, when given a starting number, ending number and step, returns a list with the numbers.

The first variation uses a fixed step value of 1, while the second variation calculates its step by sutracting the second element of the range (e_2) from the first element (e_1).

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1\ ..\ e'_2] \Rightarrow \text{range}\ e_1\ e_2\ 1} \quad (\textsc{Tr-E-Range})$$

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2 \qquad \gamma \vdash e'_3 \Rightarrow e_3 \qquad i =\ -\ e_2\ e_1}{\gamma \vdash [e'_1,\ e'_2\ ..\ e'_3] \Rightarrow \text{range}\ e_1\ e_2\ i} \quad (\textsc{Tr-E-Range2})$$

Comprehensions, similarly to ranges, rely on the function "map". A function is created with the pattern p' and the body e'_1. This function is then translated and passed as the first argument of the "map" function. The second argument of the function is the translation of the expression e'_2, which will eventually evaluate into a list.

$$\frac{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e'_1) \Rightarrow f \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1\ \text{for}\ p'\ \text{in}\ e'_2] \Rightarrow \text{map}\ f\ e_2} \quad (\textsc{Tr-E-Comprehension})$$