The Logic of Metaslang

Alessandro Coglio

March 28, 2005

DRAFT; PLEASE DO NOT DISTRIBUTE

1 Introduction

This document formally defines the logic of the Metaslang language [1], drawing ideas from [2], [3], and [4, Part II].

1.1 Notation

We define the Metaslang logic in the usual semi-formal notation consisting of naive set theory and natural language. However, it should be possible to define the Metaslang logic in axiomatic set theory or any other sufficiently expressive formal language.

The (meta-)logical notations $=$, $\forall$, $\exists$, $\wedge$, and $.\not.$ have the usual meaning.

The set-theoretic notations $\in$, $\emptyset$, $\{\ldots \mid \ldots\}$, $\{\ldots\}$, $\cup$, $\cap$, and $\subseteq$ have the usual meaning.

$\mathbf{N}$ is the set of natural numbers, i.e. $\{0, 1, 2, \ldots\}$. $\mathbf{N}_+$ is the set of positive natural numbers, i.e. $\{1, 2, 3, \ldots\}$.

If A and B are sets, $A - B$ is their difference, i.e. $\{a \in A \mid b \notin B\}$.

If A and B are sets, $A \times B$ is their cartesian product, i.e. $\{\langle a, b\rangle \mid a \in A \wedge b \in B\}$. This generalizes to $n > 2$ sets.

If A and B are sets, $A + B$ is their disjoint union, i.e. $\{\langle 0, a\rangle \mid a \in A\} \cup \{\langle 1, b\rangle \mid b \in B\}$. The "tags" 0 and 1 are always left implicit. This generalizes to $n > 2$ sets.

If A and B are sets, $A \xrightarrow{\text{p}} B$ is the set of all partial functions from A to B, i.e. $\{f \subseteq A \times B \mid \forall \langle a, b_1\rangle, \langle a, b_2\rangle \in f.\ b_1 = b_2\}$; $A \to B$ is the set of all total functions from A to B, i.e. $\{f \in A \xrightarrow{\text{p}} B \mid \forall a \in A.\ \exists b \in B.\ \langle a, b\rangle \in f\}$; and $A \hookrightarrow B$ is the set of all total injective functions from A to B, i.e. $\{f \in A \to B \mid \forall \langle a_1, b\rangle, \langle a_2, b\rangle \in f.\ a_1 = a_2\}$.

If f is a function from A to B, $\mathcal{D}(f)$ is the domain of f, i.e. $\{a \in A \mid \exists b \in B.\ \langle a, b\rangle \in f\}$.

If f is a function and $a \in \mathcal{D}(f)$, $f(a)$ denotes the unique value such that $\langle a, f(a)\rangle \in f$.

We write $f : A \xrightarrow{\text{p}} B$, $f : A \to B$, and $f : A \hookrightarrow B$ for $f \in A \xrightarrow{\text{p}} B$, $f \in A \to B$, and $f \in A \hookrightarrow B$, respectively.

If A is a set, $\mathcal{P}_\omega(A)$ is the set of all finite subsets of A, i.e. $\{S \subseteq A \mid S \text{ finite}\}$.

If A is a set, A^* is the set of all finite sequences of elements of A, i.e. $\{x_1, \ldots, x_n \mid x_1 \in A \wedge \ldots \wedge x_n \in A\}$; A^+, $A^{(*)}$, and $A^{(+)}$ are the subsets of A^* of non-empty sequences, sequences without repeated elements, and non-empty sequences without repeated elements, respectively. The empty sequence is written ϵ. The length of a sequence s is written $|s|$. When a sequence is written where a set is expected, it stands for the set of its elements.

2 Syntax

2.1 Names

We postulate the existence of an infinite set of names

$$\mathcal{N}$$

We assume that $\mathcal{N}$ contains a distinguished projection name π_i for every positive natural $i \in \mathbf{N}_+$, such that $\pi_i \neq \pi_j$ if $i \neq j$ and such that $\mathcal{N} - \{\pi_i \mid i \in \mathbf{N}_+\}$ is infinite.

2.2 Types

We inductively define the set of types as

$$\begin{aligned} \mathit{Type} = {} & \{\mathsf{Bool}\} \\ & + \{\beta \mid \beta \in \mathcal{N}\} \\ & + \{\tau(\overline{T}) \mid \tau \in \mathcal{N} \;\wedge\; \overline{T} \in \mathit{Type}^*\} \\ & + \{T_1 \to T_2 \mid T_1, T_2 \in \mathit{Type}\} \\ & + \{f_1\ T_1 \times \cdots \times f_n\ T_n \mid \overline{f} \in \mathcal{N}^{(*)} \;\wedge\; \overline{T} \in \mathit{Type}^*\} \\ & + \{c_1\ T_1 + \cdots + c_n\ T_n \mid \overline{c} \in \mathcal{N}^{(+)} \;\wedge\; \overline{T} \in \mathit{Type}^+\} \\ & + \{T|r \mid T \in \mathit{Type} \;\wedge\; r \in \mathit{Exp}\} \\ & + \{T/q \mid T \in \mathit{Type} \;\wedge\; q \in \mathit{Exp}\} \end{aligned}$$

where *Exp* is defined later.[1]

Explanation:

- There is a type Bool for boolean (i.e. truth) values.
- A name β is a type variable.
- A type instance $\tau(\overline{T})$ is obtained by combining a type name τ with zero or more argument types $\overline{T}$ that match its arity (defined later). If $\overline{T} = \epsilon$, we may abbreviate $\tau(\overline{T})$ to τ, having care to avoid confusion with type variables, which are also names.
- An arrow type $T_1 \to T_2$ is obtained by combining a domain type T_1 and a range type T_2.
- Record types $\prod_i f_i\ T_i$ (resp. sum types $\sum_i c_i\ T_i$) consist of typed fields f_i (resp. constructors c_i). Note that record types can be empty (denoted $\prod \epsilon$), while sum types cannot. All the fields (resp. constructors) of a record (resp. sum) type must be distinct names. The case of no type T_i associated to a constructor c_i in a sum type as defined in [1] is captured by T_i being $\prod \epsilon$ in the definition above: given a spec as defined in [1], one can imagine to add the empty record type where a constructor has no type, and add the empty record as argument to c_i in expressions and patterns where needed.
- Subtypes $T|r$ and quotient T/q types are obtained by combining types T with expressions r and q (meant to be suitable predicates). Subtype and quotient types create the dependency of types on expressions. Subtypes as defined above capture both restriction types and comprehension types as defined in [1]. In fact, a comprehension type can be always turned into a restriction type by re-combining the pattern and expression into a lambda expression, as mentioned in [1].

We introduce the following abbreviation

$$\textstyle\prod_i T_i \quad\longrightarrow\quad \prod_i \pi_i\ T_i$$

Thus, record types as defined above capture both record and product types as defined in [1].

[1] Types depend on expressions, which depend on types and patterns, and patterns depend on types. Thus, types, expressions, and patterns are inductively defined all together, not separately. Their definitions are presented separately only for readability.

2.3 Expressions

We inductively define the set of expressions as

$$
\begin{aligned}
\mathit{Exp} = {} & \{v \mid v \in \mathcal{N}\} \\
& + \{o[\overline{T}] \mid o \in \mathcal{N} \ \wedge\ \overline{T} \in \mathit{Type}^*\} \\
& + \{e_1\ e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\lambda v{:}T.e \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type} \ \wedge\ e \in \mathit{Exp}\} \\
& + \{e_1 \equiv e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\mathsf{if}\ e_0\ e_1\ e_2 \mid e_0, e_1, e_2 \in \mathit{Exp}\} \\
& + \{\{f_1 \leftarrow e_1 \ldots f_n \leftarrow e_n\} \mid \overline{f} \in \mathcal{N}^{(*)} \ \wedge\ \overline{e} \in \mathit{Exp}^*\} \\
& + \{e.f \mid e \in \mathit{Exp} \ \wedge\ f \in \mathcal{N}\} \\
& + \{e_1 \ll e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j \mid \textstyle\sum_i c_i\ T_i \in \mathit{Type}\} \\
& + \{\mathsf{quo}_q \mid q \in \mathit{Exp}\} \\
& + \{\mathsf{ch}_q\ e \mid q, e \in \mathit{Exp}\} \\
& + \{\mathsf{case}\ e\ \{p_1 \rightarrow e_1 \ldots p_n \rightarrow e_n\} \mid e \in \mathit{Exp} \ \wedge\ \overline{p} \in \mathit{Pat}^+ \ \wedge\ \overline{e} \in \mathit{Exp}^+\} \\
& + \{\mathsf{letr}\ \{v_1{:}T_1 \leftarrow e_1 \ldots v_n{:}T_n \leftarrow e_n\}\ e \mid \overline{v} \in \mathcal{N}^{(+)} \ \wedge\ \overline{T} \in \mathit{Type}^+ \ \wedge\ \overline{e} \in \mathit{Exp}^+ \ \wedge\ e \in \mathit{Exp}\}
\end{aligned}
$$

where Pat is defined later.

Explanation:

- A name v is a variable.
- An op(eration) instance $o[\overline{T}]$ consists of an op name o and zero or more types that instantiate the (generally, polymorphic) type of the declared op. If $\overline{T} = \epsilon$, we may abbreviate $o[\overline{T}]$ to o, having care to avoid confusion with variables, which are also names.
- A (lambda) abstraction $\lambda v{:}T.e$ consists of an argument variable v with an explicit type T and a body expression e. Even though lambda expressions as defined in [1] may have general patterns as arguments, that does not increase expressivity: one can always imagine to use a fresh variable as argument and a case expression on the fresh variable with one branch with the original pattern.
- A conditional $\mathsf{if}\ e_0\ e_1\ e_2$ consists of a condition e_0 and two branches e_1 and e_2 ("then" and "else").
- A record $\{f_i \leftarrow e_i\}_i$ consists of a sequence of pairs that associate expressions to distinct fields.
- An embedder $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j$ is decorated by a sum type and includes a constructor of that sum type. We may abbreviate $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j$ to $\mathsf{emb}\ c_j$ when the sum type is inferrable or irrelevant.
- A quotienter quo_q and a choice $\mathsf{ch}_q\ e$ are decorated by the associated predicate, which defines the quotient type.
- A recursive let $\mathsf{letr}\ \{v_i{:}T_i \leftarrow e_i\}_i\ e$ captures "let def" as defined in [1].
- An application $e_1\ e_2$, an equality $e_1 \equiv e_2$, a projection $e.f$, a record update $e_1 \ll e_2$, and a case expression $\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i$ are straightforward.

We define the following abbreviations (most of which are expressions defined in [1])

$$
\begin{array}{rcl}
\mathsf{true} & \longrightarrow & \lambda v\!:\!\mathsf{Bool}.v \equiv \lambda v\!:\!\mathsf{Bool}.v \\
\mathsf{false} & \longrightarrow & \lambda v\!:\!\mathsf{Bool}.v \equiv \lambda v\!:\!\mathsf{Bool}.\mathsf{true} \\
\neg\, e & \longrightarrow & \mathsf{if}\ e\ \mathsf{false}\ \mathsf{true} \\
e_1 \wedge e_2 & \longrightarrow & \mathsf{if}\ e_1\ e_2\ \mathsf{false} \\
e_1 \vee e_2 & \longrightarrow & \mathsf{if}\ e_1\ \mathsf{true}\ e_2 \\
e_1 \Rightarrow e_2 & \longrightarrow & \mathsf{if}\ e_1\ e_2\ \mathsf{true} \\
e_1 \Leftrightarrow e_2 & \longrightarrow & e_1 \equiv e_2 \\
 & & \text{where } e_1 \text{ and } e_2 \text{ have type } \mathsf{Bool} \text{ (defined later)} \\
e_1 \not\equiv e_2 & \longrightarrow & \neg\,(e_1 \equiv e_2) \\
\forall v\!:\!T.\ e & \longrightarrow & \lambda v\!:\!T.e \equiv \lambda v\!:\!T.\mathsf{true} \\
\exists v\!:\!T.\ e & \longrightarrow & \neg\,(\forall v\!:\!T.\ \neg\, e) \\
\exists!\, v\!:\!T.\ e & \longrightarrow & \exists v\!:\!T.\ (e \wedge \forall v'\!:\!T.\ e[v/v'] \Rightarrow v \equiv v') \\
 & & \text{where } v' \neq v\ \wedge\ v' \notin \mathcal{FV}(e)\ \wedge\ v' \notin \mathcal{CV}(e,v) \\
\mathsf{let}\ p \leftarrow e\ \mathsf{in}\ e' & \longrightarrow & \mathsf{case}\ e\ \{p \rightarrow e'\} \\
\langle \overline{e} \rangle & \longrightarrow & \{\pi_i \leftarrow e_i\}_i \\
\forall v_1\!:\!T_1,\ldots,v_n\!:\!T_n.\ e & \longrightarrow & \forall v_1\!:\!T_1.\ \ldots\ \forall v_n\!:\!T_n.\ e \\
\exists v_1\!:\!T_1,\ldots,v_n\!:\!T_n.\ e & \longrightarrow & \exists v_1\!:\!T_1.\ \ldots\ \exists v_n\!:\!T_n.\ e \\
\forall \overline{v}\!:\!\overline{T}.\ e & \longrightarrow & \forall v_1\!:\!T_1,\ldots,v_n\!:\!T_n.\ e \\
 & & \text{where } |\overline{v}| = |\overline{T}| \\
\exists \overline{v}\!:\!\overline{T}.\ e & \longrightarrow & \exists v_1\!:\!T_1,\ldots,v_n\!:\!T_n.\ e \\
 & & \text{where } |\overline{v}| = |\overline{T}|
\end{array}
$$

where substitution $\_[\_/\_]$, free variables $\mathcal{FV}$, and captured variables $\mathcal{CV}$ are defined later. It is intended that infinitely many expressions are abbreviated by true or false, one for each $v \in \mathcal{N}$; likewise, infinitely many expressions are abbreviated by $\exists!\, v : T.\ e$, one for each $v' \in \mathcal{N}$ satisfying the restrictions given above. Note that both $\forall\epsilon.\ e$ and $\exists\epsilon.\ e$ abbreviate e.

The abbreviation $\langle \overline{e} \rangle$ captures tuple displays as defined in [1]. Thus, we can regard names π_i as capturing natural literal field selectors as defined in [1].

Embedding test expressions as defined in [1] are not explicitly modeled here, either directly or as abbreviations. As explained in [1], embedding test expressions can be easily rewritten as abstractions with embedding case expressions.

Non-boolean literals and list displays as defined in [1] are not explicitly modeled here, either directly or as abbreviations, because their types are not necessarily part of every possible legal spec. They can be regarded as abbreviations for more verbose and less readable expressions obtained by applying ops defined in (library) specs for natural numbers, characters, strings, and lists.

The function $\mathcal{FV} : \mathit{Exp} \rightarrow \mathcal{P}_\omega(\mathcal{N})$ returns the free variables of an expression

$$
\begin{array}{ll}
\mathcal{FV}(v) & = \{v\} \\
\mathcal{FV}(o[\overline{T}]) & = \emptyset \\
\mathcal{FV}(e_1\ e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\lambda v\!:\!T.e) & = \mathcal{FV}(e) - \{v\} \\
\mathcal{FV}(e_1 \equiv e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\mathsf{if}\ e_0\ e_1\ e_2) & = \mathcal{FV}(e_0) \cup \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\{f_i \leftarrow e_i\}_i) & = \bigcup_i \mathcal{FV}(e_i) \\
\mathcal{FV}(e.f) & = \mathcal{FV}(e) \\
\mathcal{FV}(e_1 \ll e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\mathsf{emb}\ c_j) & = \emptyset \\
\mathcal{FV}(\mathsf{quo}_q) & = \emptyset \\
\mathcal{FV}(\mathsf{ch}_q\ e) & = \mathcal{FV}(e) \\
\mathcal{FV}(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i) & = \mathcal{FV}(e) \cup \bigcup_i (\mathcal{FV}(e_i) - \mathcal{V}(p_i)) \\
\mathcal{FV}(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e) & = (\mathcal{FV}(e) \cup \bigcup_i \mathcal{FV}(e_i)) - \overline{v}
\end{array}
$$

where $\mathcal{V}$ on patterns is defined later. The free variables of the quotient type predicate q do not contribute to the free variables of quo_q and $\mathsf{ch}_q\ e$ because, as defined later, in well-formed types and well-typed expressions those predicates have no free variables.

2.4 Patterns

We inductively define the set of patterns as

$$\begin{aligned}
\mathit{Pat} = {} & \{v{:}T \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type}\} \\
& + \{\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p \mid \textstyle\sum_i c_i\ T_i \in \mathit{Type} \ \wedge\ p \in \mathit{Pat}\} \\
& + \{\{f_1 \leftarrow p_1 \ldots f_n \leftarrow p_n\} \mid \overline{f} \in \mathcal{N}^{(*)} \ \wedge\ \overline{p} \in \mathit{Pat}^*\} \\
& + \{v{:}T \ \mathsf{as}\ p \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type} \ \wedge\ p \in \mathit{Pat}\}
\end{aligned}$$

Explanation:

- A variable pattern $v{:}T$ consists of a variable name v and an explicit type T.
- An aliased pattern $v{:}T$ as p consists of a pattern accompanied by a variable pattern.
- An embedding pattern $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p$ and a record pattern $\{f_i \leftarrow p_i\}_i$ are straightforward. We may abbreviate $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p$ to $\mathsf{emb}\ c_j\ p$ when the sum type is inferrable or irrelevant.

Literal, list, and cons patterns as defined in [1] are not explicitly modeled here, either directly or as abbreviations, because their types are not necessarily part of every possible legal spec. List and cons patterns can be regarded as abbreviations for more verbose and less readable patterns consisting of constructors defined in (library) specs for lists. Case expressions with literal patterns can be expanded into more verbose and less readable expressions that use case expressions and conditionals.

The function $\mathcal{V} : \mathit{Pat} \to \mathcal{P}_\omega(\mathcal{N})$ returns the (bound) variables in a pattern

$$\begin{aligned}
\mathcal{V}(v{:}T) &= \{v\} \\
\mathcal{V}(\mathsf{emb}\ c_j\ p) &= \mathcal{V}(p) \\
\mathcal{V}(\{f_i \leftarrow p_i\}_i) &= \textstyle\bigcup_i \mathcal{V}(p_i) \\
\mathcal{V}(v{:}T\ \mathsf{as}\ p) &= \{v\} \cup \mathcal{V}(p)
\end{aligned}$$

The function $\mathit{p2e} : \mathit{Pat} \to \mathit{Exp}$ turns a pattern into an expression

$$\begin{aligned}
\mathit{p2e}(v{:}T) &= v \\
\mathit{p2e}(\mathsf{emb}\ c_j\ p) &= \mathsf{emb}\ c_j\ \mathit{p2e}(p) \\
\mathit{p2e}(\{f_i \leftarrow p_i\}_i) &= \{f_i \leftarrow \mathit{p2e}(p_i)\}_i \\
\mathit{p2e}(v{:}T\ \mathsf{as}\ p) &= \mathit{p2e}(p)
\end{aligned}$$

Note that the extra variable v of an aliased pattern is ignored, as its purpose is just to introduce an additional binding besides those introduced by p.

The function $\mathit{pbnd} : \mathit{Pat} \to \{v{:}T \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type}\}^*$ returns the variable bindings introduced by a pattern, in a sequence

$$\begin{aligned}
\mathit{pbnd}(v{:}T) &= v{:}T \\
\mathit{pbnd}(\mathsf{emb}\ c_j\ p) &= \mathit{pbnd}(p) \\
\mathit{pbnd}(\{f_i \leftarrow p_i\}_i) &= \mathit{pbnd}(p_1), \ldots, \mathit{pbnd}(p_n) \\
\mathit{pbnd}(v{:}T\ \mathsf{as}\ p) &= v{:}T, \mathit{pbnd}(p)
\end{aligned}$$

The function $\mathit{pasm}_{\mathsf{as}} : \mathit{Pat} \to \mathit{Exp}$ returns a formula (expression) encoding the assumptions introduced by all the alias patterns that occur in a pattern

$$\begin{aligned}
\mathit{pasm}_{\mathsf{as}}(v{:}T) &= \mathsf{true} \\
\mathit{pasm}_{\mathsf{as}}(\mathsf{emb}\ c_j\ p) &= \mathit{pasm}_{\mathsf{as}}(p) \\
\mathit{pasm}_{\mathsf{as}}(\{f_i \leftarrow p_i\}_i) &= \textstyle\bigwedge_i \mathit{pasm}_{\mathsf{as}}(p_i) \\
\mathit{pasm}_{\mathsf{as}}(v{:}T\ \mathsf{as}\ p) &= \mathit{pasm}_{\mathsf{as}}(p) \wedge v \equiv \mathit{p2e}(p)
\end{aligned}$$

The function $\mathit{pasm} : \mathit{Pat} \times \mathit{Exp} \to \mathit{Exp}$ returns a formula (expression) encoding the assumption that an expression matches a pattern, along with the assumptions introduced by the alias sub-patterns

$$\mathit{pasm}(p, e) = e \equiv \mathit{p2e}(p) \wedge \mathit{pasm}_{\mathsf{as}}(p)$$

2.5 Contexts

We define the set of context elements as

$$\begin{aligned} \mathit{CxElem} = {} & \{\mathsf{ty}\ \tau\!:\!n \mid \tau \in \mathcal{N} \ \wedge\ n \in \mathbf{N}\} \\ & + \{\mathsf{op}\ o\!:\![\overline{\beta}]\ T \mid o \in \mathcal{N} \ \wedge\ \overline{\beta} \in \mathcal{N}^{(*)} \ \wedge\ T \in \mathit{Type}\} \\ & + \{\mathsf{def}\ \tau(\overline{\beta}) = T \mid \tau \in \mathcal{N} \ \wedge\ \overline{\beta} \in \mathcal{N}^{(*)} \ \wedge\ T \in \mathit{Type}\} \\ & + \{\mathsf{def}\ [\overline{\beta}]\ o = e \mid \overline{\beta} \in \mathcal{N}^{(*)} \ \wedge\ o \in \mathcal{N} \ \wedge\ e \in \mathit{Exp}\} \\ & + \{\mathsf{ax}\ [\overline{\beta}]\ e \mid \overline{\beta} \in \mathcal{N}^{(*)} \ \wedge\ e \in \mathit{Exp}\} \\ & + \{\mathsf{tvar}\ \beta \mid \beta \in \mathcal{N}\} \\ & + \{\mathsf{var}\ v\!:\!T \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type}\} \end{aligned}$$

Explanation:

- A type declaration $\mathsf{ty}\ \tau\!:\!n$ introduces a type name with an associated arity. The type variables of a type declaration as defined in [1] only serve to determine an arity and are otherwise irrelevant; thus, in the above definition we directly use the arity without type variables.
- An op(eration) declaration $\mathsf{op}\ o\!:\![\overline{\beta}]\ T$ introduces an op name with an associated type, polymorphic in the explicit type variables.
- A type definition $\mathsf{def}\ \tau(\overline{\beta}) = T$ assigns a type to a maximally generic type instance of some type name (i.e. an instance with distinct type variables as arguments to the type name). A combined type declaration and definition as defined in [1] is captured by a type declaration as defined above immediately followed by a type definition as defined above.
- An op definition $\mathsf{def}\ [\overline{\beta}]\ o = e$ assigns an expression to an op name, polymorphic in the explicit type variables. A combined op declaration and definition as defined in [1] is captured by an op declaration as defined above immediately followed by an op definition as defined above.
- An axiom $\mathsf{ax}\ [\overline{\beta}]\ e$ introduces an expression (with type Bool, as defined later), polymorphic in the explicit type variables. We may abbreviate $\mathsf{ax}\ [\epsilon]\ e$ to $\mathsf{ax}\ e$.
- A type variable declaration $\mathsf{tvar}\ \beta$ introduces a type variable.
- A variable declaration $\mathsf{var}\ v\!:\!T$ introduces a variable with a type.

We introduce the following abbreviations

$$\begin{array}{rcl} \mathsf{tvar}\ \beta_1,\ldots,\beta_n & \longrightarrow & \mathsf{tvar}\ \beta_1,\ldots,\mathsf{tvar}\ \beta_n \\ \mathsf{var}\ v_1\!:\!T_1,\ldots,v_n\!:\!T_n & \longrightarrow & \mathsf{var}\ v_1\!:\!T_1,\ldots,\mathsf{var}\ v_n\!:\!T_n \\ \mathsf{var}\ \overline{v}\!:\!\overline{T} & \longrightarrow & \mathsf{var}\ v_1,\ldots,v_nT_1,\ldots,T_n \\ & & \text{where } |\overline{v}| = |\overline{T}| \end{array}$$

We define the set of contexts as

$$\mathit{Cx} = \mathit{CxElem}^*$$

In other words, a context is a finite sequence of context elements.

The function $\mathcal{TN} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the type names declared in a context

$$\begin{aligned} \mathcal{TN}(\epsilon) &= \emptyset \\ \mathcal{TN}(\mathsf{ty}\ \tau\!:\!n, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \cup \{\tau\} \\ \mathcal{TN}(\mathsf{op}\ o\!:\![\overline{\beta}]\ T, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \\ \mathcal{TN}(\mathsf{def}\ \tau(\overline{\beta}) = T, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \\ \mathcal{TN}(\mathsf{def}\ [\overline{\beta}]\ o = e, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \\ \mathcal{TN}(\mathsf{ax}\ [\overline{\beta}]\ e, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \\ \mathcal{TN}(\mathsf{tvar}\ \beta, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \\ \mathcal{TN}(\mathsf{var}\ v\!:\!T, \mathit{cx}) &= \mathcal{TN}(\mathit{cx}) \end{aligned}$$

The function $\mathcal{ON} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the op names declared in a context

$$\begin{array}{ll}
\mathcal{ON}(\epsilon) & = \emptyset \\
\mathcal{ON}(\mathsf{ty}\ \tau{:}n,\, cx) & = \mathcal{ON}(cx) \\
\mathcal{ON}(\mathsf{op}\ o{:}[\overline{\beta}]\ T,\, cx) & = \mathcal{ON}(cx) \cup \{o\} \\
\mathcal{ON}(\mathsf{def}\ \tau(\overline{\beta}) = T,\, cx) & = \mathcal{ON}(cx) \\
\mathcal{ON}(\mathsf{def}\ [\overline{\beta}]\ o = e,\, cx) & = \mathcal{ON}(cx) \\
\mathcal{ON}(\mathsf{ax}\ [\overline{\beta}]\ e,\, cx) & = \mathcal{ON}(cx) \\
\mathcal{ON}(\mathsf{tvar}\ \beta,\, cx) & = \mathcal{ON}(cx) \\
\mathcal{ON}(\mathsf{var}\ v{:}T,\, cx) & = \mathcal{ON}(cx)
\end{array}$$

The function $\mathcal{TV} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the type variables declared in a context

$$\begin{array}{ll}
\mathcal{TV}(\epsilon) & = \emptyset \\
\mathcal{TV}(\mathsf{ty}\ \tau{:}n,\, cx) & = \mathcal{TV}(cx) \\
\mathcal{TV}(\mathsf{op}\ o{:}[\overline{\beta}]\ T,\, cx) & = \mathcal{TV}(cx) \\
\mathcal{TV}(\mathsf{def}\ \tau(\overline{\beta}) = T,\, cx) & = \mathcal{TV}(cx) \\
\mathcal{TV}(\mathsf{def}\ [\overline{\beta}]\ o = e,\, cx) & = \mathcal{TV}(cx) \\
\mathcal{TV}(\mathsf{ax}\ [\overline{\beta}]\ e,\, cx) & = \mathcal{TV}(cx) \\
\mathcal{TV}(\mathsf{tvar}\ \beta,\, cx) & = \mathcal{TV}(cx) \cup \{\beta\} \\
\mathcal{TV}(\mathsf{var}\ v{:}T,\, cx) & = \mathcal{TV}(cx)
\end{array}$$

The function $\mathcal{V} : \mathit{Cx} \to \mathcal{P}_\omega(\mathcal{N})$ returns the variables declared in a context

$$\begin{array}{ll}
\mathcal{V}(\epsilon) & = \emptyset \\
\mathcal{V}(\mathsf{ty}\ \tau{:}n,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{op}\ o{:}[\overline{\beta}]\ T,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{def}\ \tau(\overline{\beta}) = T,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{def}\ [\overline{\beta}]\ o = e,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{ax}\ [\overline{\beta}]\ e,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{tvar}\ \beta,\, cx) & = \mathcal{V}(cx) \\
\mathcal{V}(\mathsf{var}\ v{:}T,\, cx) & = \mathcal{V}(cx) \cup \{v\}
\end{array}$$

2.6 Specs

We define the set of spec(ification)s as

$$\mathit{Sp} = \{cx \in \mathit{Cx} \mid \mathcal{TV}(cx) = \mathcal{V}(cx) = \emptyset\}$$

In other words, a spec is a context without type variable declarations and variable declarations.

2.7 Occurring ops

The function $\mathcal{ON} : \mathit{Type} + \mathit{Exp} + \mathit{Pat} \to \mathcal{P}_\omega(\mathcal{N})$ returns the op names occurring in a type, expression, or pattern

$$\begin{array}{ll}
\mathcal{ON}(\mathsf{Bool}) & = \emptyset \\
\mathcal{ON}(\beta) & = \emptyset \\
\mathcal{ON}(\tau(\overline{T})) & = \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(T_1 \to T_2) & = \mathcal{ON}(T_1) \cup \mathcal{ON}(T_2) \\
\mathcal{ON}(\prod_i f_i\ T_i) & = \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(\sum_i c_i\ T_i) & = \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(T|r) & = \mathcal{ON}(T) \\
\mathcal{ON}(T/q) & = \mathcal{ON}(T)
\end{array}$$

$$\begin{array}{ll}
\mathcal{ON}(v) & = \emptyset \\
\mathcal{ON}(o[\overline{T}]) & = \{o\} \cup \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(e_1\ e_2) & = \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\
\mathcal{ON}(\lambda v\!:\!T.e) & = \mathcal{ON}(T) \cup \mathcal{ON}(e) \\
\mathcal{ON}(e_1 \equiv e_2) & = \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\
\mathcal{ON}(\mathsf{if}\ e_0\ e_1\ e_2) & = \mathcal{ON}(e_0) \cup \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\
\mathcal{ON}(\{f_i \leftarrow e_i\}_i) & = \bigcup_i \mathcal{ON}(e_i) \\
\mathcal{ON}(e.f) & = \mathcal{ON}(e) \\
\mathcal{ON}(e_1 \ll e_2) & = \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\
\mathcal{ON}(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j) & = \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(\mathsf{quo}_q) & = \mathcal{ON}(q) \\
\mathcal{ON}(\mathsf{ch}_q\ e) & = \mathcal{ON}(q) \cup \mathcal{ON}(e) \\
\mathcal{ON}(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i) & = \mathcal{ON}(e) \cup \bigcup_i (\mathcal{ON}(p_i) \cup \mathcal{ON}(e_i)) \\
\mathcal{ON}(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e) & = \mathcal{ON}(e) \cup \bigcup_i (\mathcal{ON}(T_i) \cup \mathcal{ON}(e_i))
\end{array}$$

$$\begin{array}{ll}
\mathcal{ON}(v\!:\!T) & = \mathcal{ON}(T) \\
\mathcal{ON}(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p) & = \mathcal{ON}(p) \cup \bigcup_i \mathcal{ON}(T_i) \\
\mathcal{ON}(\{f_i \leftarrow p_i\}_i) & = \bigcup_i \mathcal{ON}(p_i) \\
\mathcal{ON}(v\!:\!T\ \mathsf{as}\ p) & = \mathcal{ON}(T) \cup \mathcal{ON}(p)
\end{array}$$

2.8 Substitutions

2.8.1 Type substitutions

The function $\_[\_/\_] : \{\langle x, \overline{\beta}, \overline{S}\rangle \in (\mathit{Type}^* + \mathit{Exp} + \mathit{Pat}) \times \mathcal{N}^{(*)} \times \mathit{Type}^* \mid |\overline{\beta}| = |\overline{S}|\} \rightarrow \mathit{Type}^* + \mathit{Exp} + \mathit{Pat}$ substitutes the type variables $\overline{\beta}$ with the types $\overline{S}$ in a type (sequence), expression, or pattern x (written $x[\overline{\beta}/\overline{S}]$)

$$\begin{array}{rl}
\mathsf{Bool}[\overline{\beta}/\overline{S}] & = \mathsf{Bool} \\
\beta[\overline{\beta}/\overline{S}] & = \begin{cases} S_i & \text{if} \quad \beta = \beta_i \\ \beta & \text{otherwise} \end{cases} \\
\tau(\overline{T})[\overline{\beta}/\overline{S}] & = \tau(\overline{T}[\overline{\beta}/\overline{S}]) \\
(T_1 \rightarrow T_2)[\overline{\beta}/\overline{S}] & = T_1[\overline{\beta}/\overline{S}] \rightarrow T_2[\overline{\beta}/\overline{S}] \\
(\prod_i f_i\ T_i)[\overline{\beta}/\overline{S}] & = \prod_i f_i\ T_i[\overline{\beta}/\overline{S}] \\
(\sum_i c_i\ T_i)[\overline{\beta}/\overline{S}] & = \sum_i c_i\ T_i[\overline{\beta}/\overline{S}] \\
(T|r)[\overline{\beta}/\overline{S}] & = T[\overline{\beta}/\overline{S}]|r[\overline{\beta}/\overline{S}] \\
(T/q)[\overline{\beta}/\overline{S}] & = T[\overline{\beta}/\overline{S}]/q[\overline{\beta}/\overline{S}]
\end{array}$$

$$(T_1, \ldots, T_n)[\overline{\beta}/\overline{S}] = T_1[\overline{\beta}/\overline{S}], \ldots, T_n[\overline{\beta}/\overline{S}]$$

$$\begin{array}{rl}
v[\overline{\beta}/\overline{S}] & = v \\
o[\overline{T}][\overline{\beta}/\overline{S}] & = o[\overline{T}[\overline{\beta}/\overline{S}]] \\
(e_1\ e_2)[\overline{\beta}/\overline{S}] & = e_1[\overline{\beta}/\overline{S}]\ e_2[\overline{\beta}/\overline{S}] \\
(\lambda v\!:\!T.e)[\overline{\beta}/\overline{S}] & = \lambda v\!:\!T[\overline{\beta}/\overline{S}].e[\overline{\beta}/\overline{S}] \\
(e_1 \equiv e_2)[\overline{\beta}/\overline{S}] & = e_1[\overline{\beta}/\overline{S}] \equiv e_2[\overline{\beta}/\overline{S}] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[\overline{\beta}/\overline{S}] & = \mathsf{if}\ e_0[\overline{\beta}/\overline{S}]\ e_1[\overline{\beta}/\overline{S}]\ e_2[\overline{\beta}/\overline{S}] \\
\{f_i \leftarrow e_i\}_i[\overline{\beta}/\overline{S}] & = \{f_i \leftarrow e_i[\overline{\beta}/\overline{S}]\}_i \\
(e.f)[\overline{\beta}/\overline{S}] & = e[\overline{\beta}/\overline{S}].f \\
(e_1 \ll e_2)[\overline{\beta}/\overline{S}] & = e_1[\overline{\beta}/\overline{S}] \ll e_2[\overline{\beta}/\overline{S}] \\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\right)[\overline{\beta}/\overline{S}] & = \mathsf{emb}_{(\sum_i c_i\ T_i)[\overline{\beta}/\overline{S}]}\ c_j \\
\mathsf{quo}_q[\overline{\beta}/\overline{S}] & = \mathsf{quo}_{q[\overline{\beta}/\overline{S}]} \\
(\mathsf{ch}_q\ e)[\overline{\beta}/\overline{S}] & = \mathsf{ch}_{q[\overline{\beta}/\overline{S}]}\ e[\overline{\beta}/\overline{S}] \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[\overline{\beta}/\overline{S}] & = \mathsf{case}\ e[\overline{\beta}/\overline{S}]\ \{p_i[\overline{\beta}/\overline{S}] \rightarrow e_i[\overline{\beta}/\overline{S}]\}_i \\
(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e)[\overline{\beta}/\overline{S}] & = \mathsf{letr}\ \{v_i\!:\!T_i[\overline{\beta}/\overline{S}] \leftarrow e_i[\overline{\beta}/\overline{S}]\}_i\ e[\overline{\beta}/\overline{S}]
\end{array}$$

$$\begin{aligned}
(v\!:\!T)[\overline{\beta}/\overline{S}] &= v\!:\!T[\overline{\beta}/\overline{S}]\\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p\right)[\overline{\beta}/\overline{S}] &= \mathsf{emb}_{(\sum_i c_i\ T_i)[\overline{\beta}/\overline{S}]}\ c_j\ p[\overline{\beta}/\overline{S}]\\
\{f_i \!\leftarrow\! p_i\}_i[\overline{\beta}/\overline{S}] &= \{f_i \!\leftarrow\! p_i[\overline{\beta}/\overline{S}]\}_i\\
(v\!:\!T\ \mathsf{as}\ p)[\overline{\beta}/\overline{S}] &= v\!:\!T[\overline{\beta}/\overline{S}]\ \mathsf{as}\ p[\overline{\beta}/\overline{S}]
\end{aligned}$$

The position of a type occurrence in a type, expression, or pattern can be identified by a sequence of natural numbers that describes a path through the tree. We define the set of all possible positions as

$$\mathit{Pos} = \mathbf{N}^*$$

The relation $\_[\_/\_@\_] \rightsquigarrow \_ \subseteq (\mathit{Type}+\mathit{Exp}+\mathit{Pat}) \times \mathit{Type} \times \mathit{Type} \times \mathit{Pos} \times (\mathit{Type}+\mathit{Exp}+\mathit{Pat})$ captures that x' is the result of substituting a type S with a type S' at position ω in a type, expression, or pattern x (written $x[S/S'@\omega] \rightsquigarrow x'$)

$$\begin{aligned}
S[S/S'@\epsilon] &\rightsquigarrow S' & &\\
\tau(\overline{T})[S/S'@i,\omega] &\rightsquigarrow \tau(\ldots,T_i',\ldots) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
(T_1 \to T_2)[S/S'@1,\omega] &\rightsquigarrow (T_1' \to T_2) &\Leftarrow\quad & T_1[S/S'@\omega] \rightsquigarrow T_1'\\
(T_1 \to T_2)[S/S'@2,\omega] &\rightsquigarrow (T_1 \to T_2') &\Leftarrow\quad & T_2[S/S'@\omega] \rightsquigarrow T_2'\\
(\textstyle\prod_i f_i\ T_i)[S/S'@i,\omega] &\rightsquigarrow (\cdots \times f_i\ T_i' \times \cdots) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
(\textstyle\sum_i c_i\ T_i)[S/S'@i,\omega] &\rightsquigarrow (\cdots + c_i\ T_i' + \cdots) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
(T|r)[S/S'@0,\omega] &\rightsquigarrow (T'|r) &\Leftarrow\quad & T[S/S'@\omega] \rightsquigarrow T'\\
(T|r)[S/S'@1,\omega] &\rightsquigarrow (T|r') &\Leftarrow\quad & r[S/S'@\omega] \rightsquigarrow r'\\
(T/q)[S/S'@0,\omega] &\rightsquigarrow (T'/q) &\Leftarrow\quad & T[S/S'@\omega] \rightsquigarrow T'\\
(T/q)[S/S'@1,\omega] &\rightsquigarrow (T/q') &\Leftarrow\quad & q[S/S'@\omega] \rightsquigarrow q'
\end{aligned}$$

$$\begin{aligned}
o[\overline{T}][S/S'@i,\omega] &\rightsquigarrow o[\ldots,T_i',\ldots] &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
(e_1\ e_2)[S/S'@1,\omega] &\rightsquigarrow (e_1'\ e_2) &\Leftarrow\quad & e_1[S/S'@\omega] \rightsquigarrow e_1'\\
(e_1\ e_2)[S/S'@2,\omega] &\rightsquigarrow (e_1\ e_2') &\Leftarrow\quad & e_2[S/S'@\omega] \rightsquigarrow e_2'\\
(\lambda v\!:\!T.e)[S/S'@0,\omega] &\rightsquigarrow (\lambda v\!:\!T'.e) &\Leftarrow\quad & T[S/S'@\omega] \rightsquigarrow T'\\
(\lambda v\!:\!T.e)[S/S'@1,\omega] &\rightsquigarrow (\lambda v\!:\!T.e') &\Leftarrow\quad & e[S/S'@\omega] \rightsquigarrow e'\\
(e_1 \equiv e_2)[S/S'@1,\omega] &\rightsquigarrow (e_1' \equiv e_2) &\Leftarrow\quad & e_1[S/S'@\omega] \rightsquigarrow e_1'\\
(e_1 \equiv e_2)[S/S'@2,\omega] &\rightsquigarrow (e_1 \equiv e_2') &\Leftarrow\quad & e_2[S/S'@\omega] \rightsquigarrow e_2'\\
(\mathsf{if}\ e_0\ e_1\ e_2)[S/S'@0,\omega] &\rightsquigarrow (\mathsf{if}\ e_0'\ e_1\ e_2) &\Leftarrow\quad & e_0[S/S'@\omega] \rightsquigarrow e_0'\\
(\mathsf{if}\ e_0\ e_1\ e_2)[S/S'@1,\omega] &\rightsquigarrow (\mathsf{if}\ e_0\ e_1'\ e_2) &\Leftarrow\quad & e_1[S/S'@\omega] \rightsquigarrow e_1'\\
(\mathsf{if}\ e_0\ e_1\ e_2)[S/S'@2,\omega] &\rightsquigarrow (\mathsf{if}\ e_0\ e_1\ e_2') &\Leftarrow\quad & e_2[S/S'@\omega] \rightsquigarrow e_2'\\
\{f_i \!\leftarrow\! e_i\}_i[S/S'@i,\omega] &\rightsquigarrow \{\ldots f_i \!\leftarrow\! e_i' \ldots\} &\Leftarrow\quad & e_i[S/S'@\omega] \rightsquigarrow e_i'\\
e.f[S/S'@0,\omega] &\rightsquigarrow e'.f &\Leftarrow\quad & e[S/S'@\omega] \rightsquigarrow e'\\
(e_1 \ll e_2)[S/S'@1,\omega] &\rightsquigarrow (e_1' \ll e_2) &\Leftarrow\quad & e_1[S/S'@\omega] \rightsquigarrow e_1'\\
(e_1 \ll e_2)[S/S'@2,\omega] &\rightsquigarrow (e_1 \ll e_2') &\Leftarrow\quad & e_2[S/S'@\omega] \rightsquigarrow e_2'\\
(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j)[S/S'@i,\omega] &\rightsquigarrow (\mathsf{emb}_{\cdots+c_i\ T_i'+\cdots}\ c_j) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
\mathsf{quo}_q[S/S'@0,\omega] &\rightsquigarrow \mathsf{quo}_{q'} &\Leftarrow\quad & q[S/S'@\omega] \rightsquigarrow q'\\
(\mathsf{ch}_q\ e)[S/S'@0,\omega] &\rightsquigarrow (\mathsf{ch}_{q'}\ e) &\Leftarrow\quad & q[S/S'@\omega] \rightsquigarrow q'\\
(\mathsf{ch}_q\ e)[S/S'@1,\omega] &\rightsquigarrow (\mathsf{ch}_q\ e') &\Leftarrow\quad & e[S/S'@\omega] \rightsquigarrow e'\\
(\mathsf{case}\ e\ \{p_i \!\to\! e_i\}_i)[S/S'@0,\omega] &\rightsquigarrow (\mathsf{case}\ e'\ \{p_i \!\to\! e_i\}_i) &\Leftarrow\quad & e[S/S'@\omega] \rightsquigarrow e'\\
(\mathsf{case}\ e\ \{p_i \!\to\! e_i\}_i)[S/S'@2i-1,\omega] &\rightsquigarrow (\mathsf{case}\ e\ \{\ldots p_i' \!\to\! e_i \ldots\}) &\Leftarrow\quad & p_i[S/S'@\omega] \rightsquigarrow p_i'\\
(\mathsf{case}\ e\ \{p_i \!\to\! e_i\}_i)[S/S'@2i,\omega] &\rightsquigarrow (\mathsf{case}\ e\ \{\ldots p_i \!\to\! e_i' \ldots\}) &\Leftarrow\quad & e_i[S/S'@\omega] \rightsquigarrow e_i'\\
(\mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e)[S/S'@2i-1,\omega] &\rightsquigarrow (\mathsf{letr}\ \{\ldots v_i\!:\!T_i' \!\leftarrow\! e_i \ldots\}\ e) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
(\mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e)[S/S'@2i,\omega] &\rightsquigarrow (\mathsf{letr}\ \{\ldots v_i\!:\!T_i \!\leftarrow\! e_i' \ldots\}\ e) &\Leftarrow\quad & e_i[S/S'@\omega] \rightsquigarrow e_i'\\
(\mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e)[S/S'@0,\omega] &\rightsquigarrow (\mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e') &\Leftarrow\quad & e[S/S'@\omega] \rightsquigarrow e'
\end{aligned}$$

$$\begin{aligned}
(v\!:\!T)[S/S'@0,\omega] &\rightsquigarrow (v\!:\!T') &\Leftarrow\quad & T[S/S'@\omega] \rightsquigarrow T'\\
(\mathsf{emb}\ c_j\ p)[S/S'@0,\omega] &\rightsquigarrow (\mathsf{emb}\ c_j\ p') &\Leftarrow\quad & p[S/S'@\omega] \rightsquigarrow p'\\
(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ p)[S/S'@i,\omega] &\rightsquigarrow (\mathsf{emb}_{\cdots+c_i\ T_i'+\cdots}\ p\ ) &\Leftarrow\quad & T_i[S/S'@\omega] \rightsquigarrow T_i'\\
\{f_i \!\leftarrow\! p_i\}_i[S/S'@i,\omega] &\rightsquigarrow \{\ldots f_i \!\leftarrow\! p_i' \ldots\} &\Leftarrow\quad & p_i[S/S'@\omega] \rightsquigarrow p_i'\\
(v\!:\!T\ \mathsf{as}\ p)[S/S'@0,\omega] &\rightsquigarrow (v\!:\!T'\ \mathsf{as}\ p) &\Leftarrow\quad & T[S/S'@\omega] \rightsquigarrow T'\\
(v\!:\!T\ \mathsf{as}\ p)[S/S'@1,\omega] &\rightsquigarrow (v\!:\!T\ \mathsf{as}\ p') &\Leftarrow\quad & p[S/S'@\omega] \rightsquigarrow p'
\end{aligned}$$

2.8.2 Expression substitutions

The function $\_[\_/\_] : \mathit{Exp} \times \mathcal{N} \times \mathit{Exp} \to \mathit{Exp}$ substitutes a variable u with an expression d in an expression e (written $e[u/d]$)

$$
\begin{array}{ll}
v[u/d] & = \begin{cases} d & \text{if} \quad u = v \\ v & \text{otherwise} \end{cases} \\
o[\overline{T}][u/d] & = o[\overline{T}] \\
(e_1\ e_2)[u/d] & = e_1[u/d]\ e_2[u/d] \\
(\lambda v\!:\!T.e)[u/d] & = \begin{cases} \lambda v\!:\!T.e & \text{if} \quad u = v \\ \lambda v\!:\!T.e[u/d] & \text{otherwise} \end{cases} \\
(e_1 \equiv e_2)[u/d] & = e_1[u/d] \equiv e_2[u/d] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[u/d] & = \mathsf{if}\ e_0[u/d]\ e_1[u/d]\ e_2[u/d] \\
\{f_i \leftarrow e_i\}_i[u/d] & = \{f_i \leftarrow e_i[u/d]\}_i \\
(e.f)[u/d] & = e[u/d].f \\
(e_1 \ll e_2)[u/d] & = e_1[u/d] \ll e_2[u/d] \\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\right)[u/d] & = \mathsf{emb}_{\sum_i c_i\ T_i}\ c_j \\
\mathsf{quo}_q[u/d] & = \mathsf{quo}_q \\
(\mathsf{ch}_q\ e)[u/d] & = \mathsf{ch}_q\ e[u/d] \\
(\mathsf{case}\ e\ \{p_i \to e_i\}_i)[u/d] & = \mathsf{case}\ e[u/d]\ \{p_i \to \begin{cases} e_i & \text{if} \quad u \in \mathcal{V}(p_i) \\ e_i[u/d] & \text{otherwise} \end{cases} \}_i \\
(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e)[u/d] & = \begin{cases} \mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e & \text{if} \quad u \in \overline{v} \\ \mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i[u/d]\}_i\ e[u/d] & \text{otherwise} \end{cases}
\end{array}
$$

No substitution is performed in the quotient type predicate q because, as already mentioned, it has no free variables when it occurs in well-typed expressions.

The function $\mathcal{CV} : \mathit{Exp} \times \mathcal{N} \to \mathcal{P}_\omega(\mathcal{N})$ returns the variables that would be captured if a variable u were substituted with those variables in an expression e (i.e. all the variables bound in e at the free occurrences of u in e)

$$
\begin{array}{ll}
\mathcal{CV}(v, u) & = \emptyset \\
\mathcal{CV}(o[\overline{T}], u) & = \emptyset \\
\mathcal{CV}(e_1\ e_2, u) & = \mathcal{CV}(e_1, u) \cup \mathcal{CV}(e_2, u) \\
\mathcal{CV}(\lambda v\!:\!T.e, u) & = \begin{cases} \{v\} \cup \mathcal{CV}(e, u) & \text{if} \quad u \in \mathcal{FV}(e) \ \wedge\ u \neq v \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{CV}(e_1 \equiv e_2, u) & = \mathcal{CV}(e_1, u) \cup \mathcal{CV}(e_2, u) \\
\mathcal{CV}(\mathsf{if}\ e_0\ e_1\ e_2, u) & = \mathcal{CV}(e_0, u) \cup \mathcal{CV}(e_1, u) \cup \mathcal{CV}(e_2, u) \\
\mathcal{CV}(\{f_i \leftarrow e_i\}_i, u) & = \bigcup_i \mathcal{CV}(e_i, u) \\
\mathcal{CV}(e.f, u) & = \mathcal{CV}(e, u) \\
\mathcal{CV}(e_1 \ll e_2, u) & = \mathcal{CV}(e_1, u) \cup \mathcal{CV}(e_2, u) \\
\mathcal{CV}(\mathsf{emb}\ c_j, u) & = \emptyset \\
\mathcal{CV}(\mathsf{quo}_q, u) & = \emptyset \\
\mathcal{CV}(\mathsf{ch}_q\ e, u) & = \mathcal{CV}(e, u) \\
\mathcal{CV}(\mathsf{case}\ e\ \{p_i \to e_i\}_i, u) & = \mathcal{CV}(e, u) \cup \bigcup_i \begin{cases} \mathcal{V}(p_i) \cup \mathcal{CV}(e_i, u) & \text{if} \quad u \in \mathcal{FV}(e_i) - \mathcal{V}(p_i) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{CV}(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e, u) & = \begin{cases} \overline{v} \cup \mathcal{CV}(e, u) \cup \bigcup_i \mathcal{CV}(e_i, u) & \text{if} \quad u \in (\mathcal{FV}(e) \cup \bigcup_i \mathcal{FV}(e_i)) - \overline{v} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

The relation $\mathit{OKsbs} \subseteq \mathit{Exp} \times \mathcal{N} \times \mathit{Exp}$ captures the condition that the substitution $e[u/d]$ causes no free variables in d to be captured

$$\mathit{OKsbs}(e, u, d) \ \Leftrightarrow\ \mathcal{FV}(d) \cap \mathcal{CV}(e, u) = \emptyset$$

Similarly to type occurrences in types, expression, and patterns, the position of an expression occurrence in an expression can be identified by a sequence of natural numbers that describes a path through the tree, i.e. an element of Pos (defined earlier).

The relation $\_[\_/\_@\_] \rightsquigarrow \_ \subseteq \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \times \mathit{Pos} \times \mathit{Exp}$ captures that e' is the result of substituting an expression d with an expression d' at position ω in an expression e (written $e[d/d'@\omega] \rightsquigarrow e'$)

$$\begin{array}{rcl}
d[d/d'@\epsilon] \rightsquigarrow d' & & \\
(e_1\ e_2)[d/d'@1,\omega] \rightsquigarrow (e_1'\ e_2) & \Leftarrow & e_1[d/d'@\omega] \rightsquigarrow e_1' \\
(e_1\ e_2)[d/d'@2,\omega] \rightsquigarrow (e_1\ e_2') & \Leftarrow & e_2[d/d'@\omega] \rightsquigarrow e_2' \\
(\lambda v\!:\!T.e)[d/d'@0,\omega] \rightsquigarrow (\lambda v\!:\!T.e') & \Leftarrow & e[d/d'@\omega] \rightsquigarrow e' \\
(e_1 \equiv e_2)[d/d'@1,\omega] \rightsquigarrow (e_1' \equiv e_2) & \Leftarrow & e_1[d/d'@\omega] \rightsquigarrow e_1' \\
(e_1 \equiv e_2)[d/d'@2,\omega] \rightsquigarrow (e_1 \equiv e_2') & \Leftarrow & e_2[d/d'@\omega] \rightsquigarrow e_2' \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d/d'@0,\omega] \rightsquigarrow (\mathsf{if}\ e_0'\ e_1\ e_2) & \Leftarrow & e_0[d/d'@\omega] \rightsquigarrow e_0' \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d/d'@1,\omega] \rightsquigarrow (\mathsf{if}\ e_0\ e_1'\ e_2) & \Leftarrow & e_1[d/d'@\omega] \rightsquigarrow e_1' \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d/d'@2,\omega] \rightsquigarrow (\mathsf{if}\ e_0\ e_1\ e_2') & \Leftarrow & e_2[d/d'@\omega] \rightsquigarrow e_2' \\
\{f_i \leftarrow e_i\}_i[d/d'@i,\omega] \rightsquigarrow \{\ldots f_i \leftarrow e_i' \ldots\} & \Leftarrow & e_i[d/d'@\omega] \rightsquigarrow e_i' \\
e.f[d/d'@0,\omega] \rightsquigarrow e'.f & \Leftarrow & e[d/d'@\omega] \rightsquigarrow e' \\
(e_1 \ll e_2)[d/d'@1,\omega] \rightsquigarrow (e_1' \ll e_2) & \Leftarrow & e_1[d/d'@\omega] \rightsquigarrow e_1' \\
(e_1 \ll e_2)[d/d'@2,\omega] \rightsquigarrow (e_1 \ll e_2') & \Leftarrow & e_2[d/d'@\omega] \rightsquigarrow e_2' \\
(\mathsf{ch}_q\ e)[d/d'@0,\omega] \rightsquigarrow (\mathsf{ch}_q\ e') & \Leftarrow & e[d/d'@\omega] \rightsquigarrow e' \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[d/d'@0,\omega] \rightsquigarrow (\mathsf{case}\ e'\ \{p_i \rightarrow e_i\}_i) & \Leftarrow & e[d/d'@\omega] \rightsquigarrow e' \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[d/d'@i,\omega] \rightsquigarrow (\mathsf{case}\ e\ \{\ldots p_i \rightarrow e_i' \ldots\}) & \Leftarrow & e_i[d/d'@\omega] \rightsquigarrow e_i' \\
(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e)[d/d'@0,\omega] \rightsquigarrow (\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e') & \Leftarrow & e[d/d'@\omega] \rightsquigarrow e' \\
(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e)[d/d'@i,\omega] \rightsquigarrow (\mathsf{letr}\ \{\ldots v_i\!:\!T_i \leftarrow e_i' \ldots\}\ e) & \Leftarrow & e_i[d/d'@\omega] \rightsquigarrow e_i'
\end{array}$$

The relation $\mathit{posIn} \subseteq \mathit{Pos} \times \mathit{Exp}$ captures legal positions in expressions

$$\mathit{posIn}(\omega, e) \ \Leftrightarrow\ \exists e', d, d'.\ \ e[d/d'@\omega] \rightsquigarrow e'$$

The function $\mathcal{CV} : \{\langle e, \omega\rangle \in \mathit{Exp} \times \mathit{Pos} \mid \mathit{posIn}(\omega, e)\} \rightarrow \mathcal{P}_\omega(\mathcal{N})$ returns the variables that would be captured if they were substituted in an expression e at position ω (i.e. all the variables bound in e at position ω)

$$\begin{array}{ll}
\mathcal{CV}(e, \epsilon) & = \emptyset \\
\mathcal{CV}(e_1\ e_2, (1, \omega)) & = \mathcal{CV}(e_1, \omega) \\
\mathcal{CV}(e_1\ e_2, (2, \omega)) & = \mathcal{CV}(e_2, \omega) \\
\mathcal{CV}(\lambda v\!:\!T.e, (0, \omega)) & = \{v\} \cup \mathcal{CV}(e, \omega) \\
\mathcal{CV}(e_1 \equiv e_2, (1, \omega)) & = \mathcal{CV}(e_1, \omega) \\
\mathcal{CV}(e_1 \equiv e_2, (2, \omega)) & = \mathcal{CV}(e_2, \omega) \\
\mathcal{CV}(\mathsf{if}\ e_0\ e_1\ e_2, (0, \omega)) & = \mathcal{CV}(e_0, \omega) \\
\mathcal{CV}(\mathsf{if}\ e_0\ e_1\ e_2, (1, \omega)) & = \mathcal{CV}(e_1, \omega) \\
\mathcal{CV}(\mathsf{if}\ e_0\ e_1\ e_2, (2, \omega)) & = \mathcal{CV}(e_2, \omega) \\
\mathcal{CV}(\{f_i \leftarrow e_i\}_i, (i, \omega)) & = \mathcal{CV}(e_i, \omega) \\
\mathcal{CV}(e.f, (0, \omega)) & = \mathcal{CV}(e, \omega) \\
\mathcal{CV}(e_1 \ll e_2, (1, \omega)) & = \mathcal{CV}(e_1, \omega) \\
\mathcal{CV}(e_1 \ll e_2, (2, \omega)) & = \mathcal{CV}(e_2, \omega) \\
\mathcal{CV}(\mathsf{ch}_q\ e, (0, \omega)) & = \mathcal{CV}(e, \omega) \\
\mathcal{CV}(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i, (0, \omega)) & = \mathcal{CV}(e, \omega) \\
\mathcal{CV}(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i, (i, \omega)) & = \mathcal{V}(p_i) \cup \mathcal{CV}(e_i, \omega) \\
\mathcal{CV}(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e, (0, \omega)) & = \overline{v} \cup \mathcal{CV}(e, \omega) \\
\mathcal{CV}(\mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e, (i, \omega)) & = \overline{v} \cup \mathcal{CV}(e_i, \omega)
\end{array}$$

The relation $\mathit{OKsbs} \subseteq \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \times \mathit{Pos}$ captures the condition that the substitution $e[d/d'@\omega] \rightsquigarrow e'$ is defined (for some e'), affects no free variables in d that are bound in e, and causes no free variables in d' to be captured

$$\mathit{OKsbs}(e, d, d', \omega) \ \Leftrightarrow\ \exists e'.\ \ e[d/d'@\omega] \rightsquigarrow e' \ \wedge\ \mathcal{FV}(d) \cap \mathcal{CV}(e, \omega) = \emptyset \ \wedge\ \mathcal{FV}(d') \cap \mathcal{CV}(e, \omega) = \emptyset$$

2.8.3 Pattern substitutions

The function $\_[\_/\_] : Pat \times \mathcal{N} \times \mathcal{N} \to Pat$ substitutes a variable u with a variable u' in a pattern p (written $p[u/u']$)

$$\begin{array}{ll}
(v\!:\!T)[u/u'] & = \begin{cases} u'\!:\!T & \text{if} \quad u = v \\ v\!:\!T & \text{otherwise} \end{cases} \\
(\mathsf{emb}\ c_j\ p)[u/u'] & = \mathsf{emb}\ c_j\ p[u/u'] \\
\{f_i \!\leftarrow\! p_i\}_i[u/u'] & = \{f_i \!\leftarrow\! p_i[u/u']\}_i \\
(v\!:\!T\ \mathsf{as}\ p)[u/u'] & = \begin{cases} u'\!:\!T\ \mathsf{as}\ p[u/u'] & \text{if} \quad u = v \\ v\!:\!T\ \mathsf{as}\ p[u/u'] & \text{otherwise} \end{cases}
\end{array}$$

3 Proof theory

The proof theory of the Metaslang logic includes not only rules to derive formulas (theorems), but also rules to derive typing judgements, type equivalences, and other judgements. The rules to derive such judgements are mutually recursive; even though they are presented separately in the following subsections, the rules are inductively defined all together.

3.1 Well-formed contexts

We define a unary relation $\vdash \_ : \textsc{context} \subseteq Cx$ to capture well-formed contexts as

$$\frac{}{\vdash \epsilon : \textsc{context}} \quad (\textsc{cxMT})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \tau \notin \mathcal{TN}(cx)\end{array}}{\vdash cx, \mathsf{ty}\ \tau\!:\!n : \textsc{context}} \quad (\textsc{cxTdec})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ o \notin \mathcal{ON}(cx) \\ cx, \mathsf{tvar}\ \overline{\beta} \vdash T : \textsc{type}\end{array}}{\vdash cx, \mathsf{op}\ o\!:\![\overline{\beta}]\ T : \textsc{context}} \quad (\textsc{cxOdec})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{ty}\ \tau\!:\!n \in cx \\ \mathsf{def}\ \tau \ldots \notin cx \\ cx, \mathsf{tvar}\ \overline{\beta} \vdash T : \textsc{type} \\ |\overline{\beta}| = n\end{array}}{\vdash cx, \mathsf{def}\ \tau(\overline{\beta}) = T : \textsc{context}} \quad (\textsc{cxTdef})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{op}\ o\!:\![\overline{\beta}]\ T \in cx \\ \mathsf{def} \ldots o \ldots \notin cx \\ cx, \mathsf{tvar}\ \overline{\beta}' \vdash \exists!\, v\!:\!T[\overline{\beta}/\overline{\beta}'].\ v \equiv e \\ o \notin \mathcal{ON}(e)\end{array}}{\vdash cx, \mathsf{def}\ [\overline{\beta}']\ o = e[v/o[\overline{\beta}']] : \textsc{context}} \quad (\textsc{cxOdef})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ cx, \mathsf{tvar}\ \overline{\beta} \vdash e : \mathsf{Bool}\end{array}}{\vdash cx, \mathsf{ax}\ [\overline{\beta}]\ e : \textsc{context}} \quad (\textsc{cxAx})$$

$$\frac{\begin{array}{c}\vdash cx : \text{CONTEXT} \\ \beta \notin \mathcal{TV}(cx)\end{array}}{\vdash cx, \mathsf{tvar}\ \beta : \text{CONTEXT}} \quad (\text{CXTVDEC})$$

$$\frac{\begin{array}{c}\vdash cx : \text{CONTEXT} \\ v \notin \mathcal{V}(cx) \\ cx \vdash T : \text{TYPE}\end{array}}{\vdash cx, \mathsf{var}\ v\!:\!T : \text{CONTEXT}} \quad (\text{CXVDEC})$$

Eplanation:

- The empty context ϵ is well-formed. All other rules add context elements to well-formed contexts. Thus, a well-formed context is constructed incrementally starting with the empty one and adding suitable elements.
- A type declaration $\mathsf{ty}\ \tau\!:\!n$ can be added to cx if τ is not already declared in cx.
- An op declaration $\mathsf{op}\ o\!:\![\overline{\beta}]\ T$ can be added to cx if o is not already declared in cx. The op's type T must be well-formed (defined later) in cx extended with the type variables $\overline{\beta}$, which must be distinct. Note that we do not require that all type variables in T are among $\overline{\beta}$, because there is no need for that restriction. However, since a well-formed spec has no type variable declarations, an op declaration in a well-formed spec automatically satisfies the restriction.
- A type definition $\mathsf{def}\ \tau(\overline{\beta}) = T$ can be added to cx if τ is already declared but not already defined in cx. The defining type T must be well-formed in cx extended with the type variables $\overline{\beta}$, which must be distinct and whose number must match the arity of τ. Similarly to op declarations, we do not require that all type variables in T are among $\overline{\beta}$, but such a restriction is automatically satisfied in a well-formed spec. Note that we allow vacuous type definitions such as $\mathsf{def}\ \tau(\epsilon) = \tau$ or $\mathsf{def}\ \tau(\epsilon) = \tau', \mathsf{def}\ \tau'(\epsilon) = \tau$, as well as other (mutually) recursive definitions that do not uniquely pin down the defined type such as the usual definition of lists; uniqueness can be enforced by suitable axioms (e.g. induction on lists). Unlike [1], there is no implicit assumption of (mutually) recursively defined types having least fixpoint semantics because there is no general way to generate implicit axioms expressing least fixpoint semantics in the Metaslang type system.
- An op definition for o can be added to cx if o is already declared but not already defined in cx. It is allowed to use different type variables $\overline{\beta}'$ from the type variables $\overline{\beta}$ used in the declaration of o, as long as they are also distinct and are the same number (i.e. it is an injective renaming); accordingly, the type T of o becomes $T[\overline{\beta}/\overline{\beta}']$. Of course, it is possible that $\overline{\beta}' = \overline{\beta}$. The defining expression of o must be such that there is a unique solution to the equation obtained by replacing o with some variable v in the defining equation of o; turning "replacement of o with v" around, the equation is $v \equiv e$ and the defining expression of o is $e[v/o[\overline{\beta}']]$. The uniqueness of the solution is expressed as a theorem (defined later) in cx extended with the type variables $\overline{\beta}'$.
- An formula e can be added to cx as an axiom if e has type Bool (defined later). In general, the axiom may be polymorphic in (distinct) type variables $\overline{\beta}$. As in other cases, all type variables in e are automatically in $\overline{\beta}$ in well-formed specs, but that is not required in well-formed contexts in general.
- A type variable declaration $\mathsf{tvar}\ \beta$ can be added to cx if β is not already declared in cx.
- A variable declaration $\mathsf{var}\ v\!:\!T$ can be added to cx if v is not already declared in cx and T is well-formed type in cx.

3.2 Well-formed specs

We define a unary relation $\vdash \_ : \textsc{spec} \subseteq Sp$ to capture well-formed specs as

$$\frac{\vdash sp : \textsc{context}}{\vdash sp : \textsc{spec}} \quad (\textsc{Spec})$$

3.3 Well-formed types

We define a binary relation $\_ \vdash \_ : \textsc{type} \subseteq Cx \times Type$ to capture well-formed types as

$$\frac{\vdash cx : \textsc{context}}{cx \vdash \mathsf{Bool} : \textsc{type}} \quad (\textsc{tyBool})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context}\\ \beta \in \mathcal{TV}(cx)\end{array}}{cx \vdash \beta : \textsc{type}} \quad (\textsc{tyVar})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context}\\ \mathsf{ty}\ \tau : n \in cx\\ |\overline{T}| = n\\ \forall i.\ \ cx \vdash T_i : \textsc{type}\end{array}}{cx \vdash \tau(\overline{T}) : \textsc{type}} \quad (\textsc{tyInst})$$

$$\frac{\begin{array}{c}cx \vdash T_1 : \textsc{type}\\ cx \vdash T_2 : \textsc{type}\end{array}}{cx \vdash T_1 \rightarrow T_2 : \textsc{type}} \quad (\textsc{tyArr})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context}\\ \forall i.\ \ cx \vdash T_i : \textsc{type}\end{array}}{cx \vdash \prod_i f_i\ T_i : \textsc{type}} \quad (\textsc{tyRec})$$

$$\frac{\forall i.\ \ cx \vdash T_i : \textsc{type}}{cx \vdash \sum_i c_i\ T_i : \textsc{type}} \quad (\textsc{tySum})$$

$$\frac{\begin{array}{c}cx \vdash r : T \rightarrow \mathsf{Bool}\\ \mathcal{FV}(r) = \emptyset\end{array}}{cx \vdash T|r : \textsc{type}} \quad (\textsc{tySub})$$

$$\frac{\begin{array}{c}cx \vdash \forall v\!:\!T.\ q\ \langle v, v\rangle\\ cx \vdash \forall v'\!:\!T, v''\!:\!T.\ q\ \langle v', v''\rangle \Rightarrow q\ \langle v'', v'\rangle\\ cx \vdash \forall v_1\!:\!T, v_2\!:\!T, v_3\!:\!T.\ q\ \langle v_1, v_2\rangle \wedge q\ \langle v_2, v_3\rangle \Rightarrow q\ \langle v_1, v_3\rangle\\ v' \neq v''\ \ \wedge\ \ v_1 \neq v_2\ \ \wedge\ \ v_2 \neq v_3\ \ \wedge\ \ v_1 \neq v_3\\ \mathcal{FV}(q) = \emptyset\end{array}}{cx \vdash T/q : \textsc{type}} \quad (\textsc{tyQuot})$$

Explanation:

- The type Bool is well-formed in any well-formed context.
- A type variable is a well-formed type in any well-formed context that declares it.

- A type instance $\tau(\overline{T})$ is well-formed in any well-formed context cx that declares τ if the argument types are well-formed in cx and their number matches the arity of τ.
- The rules for arrow, record, and sum types are straightforward. Note that in tyArr and tySum we do not explicitly require cx to be well-formed because the fact that a type is well-formed in a context implies that the context is well-formed (as proved later). However, the condition is explicit in tyRec because a record type can have zero components (unlike a sum type that always has at least one component).
- For subtypes, we require the predicate to have type $T \rightarrow \mathsf{Bool}$, which implies that T is a well-formed type (as proved later). We also require that r has no free variables, otherwise we would implicitly have a form of dependent types.
- For quotient types, we require the predicate to be an equivalence relation, i.e. that reflexivity, symmetry, and transitivity are theorems, which implies that T and cx are well-formed, that q has type $T \times T \rightarrow \mathsf{Bool}$, etc. (as proved later). The condition that the variables v', v'', etc. are distinct is important: without it, the symmetry and transitivity requirements would effectively disappear (because the corresponding formulas would be trivially provable), thus only requiring the binary relation to be reflexive.

3.4 Type equivalence

We define a ternary relation $\_ \vdash \_ \approx \_ \subseteq \mathit{Cx} \times \mathit{Type} \times \mathit{Type}$ to capture type equivalence as

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{def}\ \tau(\overline{\beta}) = T \in cx \\ \forall i.\ \ cx \vdash T_i : \textsc{type}\end{array}}{cx \vdash \tau(\overline{T}) \approx T[\overline{\beta}/\overline{T}]} \quad (\textsc{teDef})$$

$$\frac{cx \vdash T : \textsc{type}}{cx \vdash T \approx T} \quad (\textsc{teRefl})$$

$$\frac{cx \vdash T_1 \approx T_2}{cx \vdash T_2 \approx T_1} \quad (\textsc{teSymm})$$

$$\frac{\begin{array}{c}cx \vdash T_1 \approx T_2 \\ cx \vdash T_2 \approx T_3\end{array}}{cx \vdash T_1 \approx T_3} \quad (\textsc{teTrans})$$

$$\frac{\begin{array}{c}cx \vdash T : \textsc{type} \\ cx \vdash T_1 \approx T_2 \\ T[T_1/T_2@\omega] \rightsquigarrow T'\end{array}}{cx \vdash T \approx T'} \quad (\textsc{teSubst})$$

$$\frac{\begin{array}{c}cx \vdash \prod_i f_i\ T_i : \textsc{type} \\ P : \{1, \ldots, n\} \hookrightarrow \{1, \ldots, n\}\end{array}}{cx \vdash \prod_i f_i\ T_i \approx \prod_i f_{P(i)}\ T_{P(i)}} \quad (\textsc{teRecOrd})$$

$$\frac{\begin{array}{c}cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ P : \{1, \ldots, n\} \hookrightarrow \{1, \ldots, n\}\end{array}}{cx \vdash \sum_i c_i\ T_i \approx \sum_i c_{P(i)}\ T_{P(i)}} \quad (\textsc{teSumOrd})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \text{TYPE} \\ cx \vdash T|r' : \text{TYPE} \\ cx \vdash r \equiv r' \end{array}}{cx \vdash T|r \approx T|r'} \quad (\text{teSubPred})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \text{TYPE} \\ cx \vdash T/q' : \text{TYPE} \\ cx \vdash q \equiv q' \end{array}}{cx \vdash T/q \approx T/q'} \quad (\text{teQuotPred})$$

Explanation:

- A type definition introduces type equivalences, one for each instance of the defining equation.
- Type equivalence is indeed an equivalence, i.e. reflexive, symmetric, and transitive.
- Type equivalence is a congruence with respect to syntactic (meta-)operations to construct types in the Metaslang type system, namely type instantiations, arrow types, record types, sum types, subtypes, and quotient types. This is captured by rule teSubst, which states that substituting equivalent types maintains equivalent types.
- Equal subtype or quotient predicates give rise to equivalent subtypes or quotient types.
- The order of the components of a record or sum type is unimportant, because any permutation of the components yields equivalent types. In the rules, the permutation is captured by a bijective function P on the record or sum indices $\{1, \ldots, n\}$ (the rules explicitly say that P is injective only, but since domain and codomain are finite and equal, it follows that P is also surjective, hence bijective).

3.5 Subtyping

We define a quaternary relation $\_ \vdash \_ \prec_{\_} \_ \subseteq \mathit{Cx} \times \mathit{Type} \times \mathit{Exp} \times \mathit{Type}$ to capture subtyping as

$$\frac{cx \vdash T|r : \text{TYPE}}{cx \vdash T|r \prec_r T} \quad (\text{stSub})$$

$$\frac{\begin{array}{c} cx \vdash T : \text{TYPE} \\ r = \lambda v\!:\!T.\mathsf{true} \end{array}}{cx \vdash T \prec_r T} \quad (\text{stRefl})$$

$$\frac{\begin{array}{c} cx \vdash T : \text{TYPE} \\ cx \vdash T_1 \prec_r T_2 \\ v \neq v' \\ r' = \lambda v\!:\!T \to T_2 . \forall v'\!:\!T.\ r\ (v\ v') \end{array}}{cx \vdash T \to T_1 \prec_{r'} T \to T_2} \quad (\text{stArr})$$

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \text{TYPE} \\ \forall i.\ \ cx \vdash T_i \prec_{r_i} T_i' \\ r = \lambda v\!:\!\prod_i f_i\ T_i' . \bigwedge_i r_i\ v.f_i \end{array}}{cx \vdash \prod_i f_i\ T_i \prec_r \prod_i f_i\ T_i'} \quad (\text{stRec})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ \forall i.\ \ cx \vdash T_i \prec_{r_i} T'_i \\ r = \lambda v{:}\sum_i c_i\ T_i.\mathsf{case}\ v\ \{\mathsf{emb}\ c_i\ (v_i{:}T'_i) \rightarrow r_i\ v_i\}_i \end{array}}{cx \vdash \sum_i c_i\ T_i \prec_r \sum_i c_i\ T'_i} \quad (\textsc{stSum})$$

$$\frac{\begin{array}{c} cx \vdash T_1 \approx T'_1 \\ cx \vdash T_2 \approx T'_2 \\ cx \vdash T_1 \prec_r T_2 \end{array}}{cx \vdash T'_1 \prec_r T'_2} \quad (\textsc{stTE})$$

Explanation:

- Unsurprisingly, an explicit subtype $T|r$ is a subtype of T, with r being the predicate over the supertype T that identifies the values that are also in the subtype $T|r$.
- Subtyping is reflexive, i.e. a (well-formed) type T is a subtype of itself and the associated subtype predicate is always true.
- Arrow types are monotone in their range types with respect to subtyping. The associated predicate holds when all the values of the function satisfy the predicate associated to the range subtype. Note that the domain must be the same; while domain contravariance is used in some type systems with subtypes, it would violate extensionality (see explanation in [3]).
- Both record and sum types are monotone in their component types with respect to subtyping. The record subtype predicate holds when all the component subtype predicates hold on the record components. The sum subtype predicate holds when the appropriate component subtype predicate holds on the value of the sum type (a case expression is used to discriminate the component).
- Rule stTE says that, as expected, type equivalence is a congruence with respect to subtyping judgements.

We do not need a transitivity rule for subtyping. As defined later, subtyping judgements are only used to assign types to expressions, e.g. to assign a supertype to an expression of a subtype. So, instead of using transitivity of subtyping, rules for well-typed expressions can be applied multiple times, achieving the same effect.

3.6 Well-typed expressions

We define a ternary relation $\_ \vdash \_ : \_ \subseteq Cx \times Exp \times Type$ to capture well-typed expressions as

$$\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ \mathsf{var}\ v{:}T \in cx \end{array}}{cx \vdash v : T} \quad (\textsc{exVar})$$

$$\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ \mathsf{op}\ o{:}[\overline{\beta}]\ T \in cx \\ \forall i.\ \ cx \vdash T_i : \textsc{type} \end{array}}{cx \vdash o[\overline{T}] : T[\overline{\beta}/\overline{T}]} \quad (\textsc{exOp})$$

$$\frac{\begin{array}{c} cx \vdash e_1 : T_1 \rightarrow T_2 \\ cx \vdash e_2 : T_1 \end{array}}{cx \vdash e_1\ e_2 : T_2} \quad (\textsc{exApp})$$

$$\frac{cx, \mathsf{var}\ v\!:\!T \vdash e : T'}{cx \vdash \lambda v\!:\!T.e : T \rightarrow T'} \quad (\textsc{exAbs})$$

$$\frac{\begin{array}{c} cx \vdash e_1 : T \\ cx \vdash e_2 : T \end{array}}{cx \vdash e_1 \equiv e_2 : \mathsf{Bool}} \quad (\textsc{exEq})$$

$$\frac{\begin{array}{c} cx \vdash e_0 : \mathsf{Bool} \\ cx, \mathsf{ax}\ e_0 \vdash e_1 : T \\ cx, \mathsf{ax}\ \neg\ e_0 \vdash e_2 : T \end{array}}{cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 : T} \quad (\textsc{exIf})$$

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \textsc{type} \\ \forall i.\ \ cx \vdash e_i : T_i \end{array}}{cx \vdash \{f_i \leftarrow e_i\}_i : \prod_i f_i\ T_i} \quad (\textsc{exRec})$$

$$\frac{cx \vdash e : \prod_i f_i\ T_i}{cx \vdash e.f_i : T_i} \quad (\textsc{exProj})$$

$$\frac{\begin{array}{c} cx \vdash e_1 : \prod_i f_i\ T_i \times \prod_j f'_j\ T'_j \\ cx \vdash e_2 : \prod_i f_i\ T_i \times \prod_j f''_j\ T''_j \\ \overline{f}' \cap \overline{f}'' = \emptyset \end{array}}{cx \vdash e_1 \ll e_2 : \prod_i f_i\ T_i \times \prod_j f'_j\ T'_j \times \prod_j f''_j\ T''_j} \quad (\textsc{exRecUpd})$$

$$\frac{cx \vdash \sum_i c_i\ T_i : \textsc{type}}{cx \vdash \mathsf{emb}\ c_j : T_j \rightarrow \sum_i c_i\ T_i} \quad (\textsc{exEmbed})$$

$$\frac{cx \vdash T/q : \textsc{type}}{cx \vdash \mathsf{quo}_q : T \rightarrow T/q} \quad (\textsc{exQuot})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ cx \vdash e : T \rightarrow T' \\ cx \vdash \forall v\!:\!T.\ \forall v'\!:\!T.\ q\ \langle v, v' \rangle \Rightarrow e\ v \equiv e\ v' \\ v \neq v' \\ v, v' \notin \mathcal{FV}(e) \end{array}}{cx \vdash \mathsf{ch}_q\ e : T/q \rightarrow T'} \quad (\textsc{exChoose})$$

$$\frac{\begin{array}{c} cx \vdash e : T \\ \forall i.\ \ cx \vdash p_i : T \\ cx \vdash \bigvee_i \exists pbnd(p_i).\ pasm(p_i, e) \\ \forall i.\ \ cx_i^- = \mathsf{ax}\ \bigwedge_{j<i} \forall pbnd(p_j).\ \neg\ pasm(p_j, e) \\ \forall i.\ \ cx_i^+ = \mathsf{var}\ pbnd(p_i), \mathsf{ax}\ pasm(p_i, e) \\ \forall i.\ \ cx, cx_i^-, cx_i^+ \vdash e_i : T' \end{array}}{cx \vdash \mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i : T'} \quad (\textsc{exCase})$$

$$\frac{\begin{array}{c} cx \vdash \exists!\, v\!:\!\prod_i T_i.\ \bigwedge_i v.\pi_i \equiv e_i[v_1/v.\pi_1]\cdots[v_n/v.\pi_n] \\ v \notin \overline{v} \cup \bigcup_i \mathcal{FV}(e_i) \\ cx, \mathsf{var}\ \overline{v}\!:\!\overline{T} \vdash e : T \end{array}}{cx \vdash \mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e : T} \quad (\text{exLetRec})$$

$$\frac{\begin{array}{c} cx \vdash e : T \\ cx \vdash T \prec_r T' \end{array}}{cx \vdash e : T'} \quad (\text{exSuper})$$

$$\frac{\begin{array}{c} cx \vdash e : T' \\ cx \vdash T \prec_r T' \\ cx \vdash r\ e \end{array}}{cx \vdash e : T} \quad (\text{exSub})$$

$$\frac{\begin{array}{c} cx \vdash \lambda v\!:\!T.e : T' \\ v' \notin \mathcal{FV}(e) \cup \mathcal{CV}(e,v) \end{array}}{cx \vdash \lambda v'\!:\!T.e[v/v'] : T'} \quad (\text{exAbsAlpha})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{case}\ e\ \{\ldots p_i \rightarrow e_i \ldots\} : T \\ v \in \mathcal{V}(p_i) \\ v' \notin \mathcal{V}(p_i) \cup \mathcal{FV}(e_i) \cup \mathcal{CV}(e_i,v) \end{array}}{cx \vdash \mathsf{case}\ e\ \{\ldots p_i[v/v'] \rightarrow e_i[v/v'] \ldots\} : T} \quad (\text{exCaseAlpha})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{letr}\ \{v_i\!:\!T_i \leftarrow e_i\}_i\ e : T \\ v'_j \notin \overline{v} \cup \mathcal{FV}(e) \cup \mathcal{CV}(e,v_j) \cup \bigcup_i(\mathcal{FV}(e_i) \cup \mathcal{CV}(e_i,v_j)) \\ \forall i \neq j.\ \ v'_i = v_i \end{array}}{cx \vdash \mathsf{letr}\ \{v'_i\!:\!T_i \leftarrow e_i[v_j/v'_j]\}_i\ e[v_j/v'_j] : T} \quad (\text{exLetRecAlpha})$$

Explanation:

- An op o declared in a well-formed context can be instantiated via well-formed types $\overline{T}$ whose number matches the number of type variables $\overline{\beta}$. The result is a well-formed expression whose type is obtained by substituting $\overline{\beta}$ with $\overline{T}$ in the declared type T of o.
- A variable v declared in a well-formed context is a well-typed expression with the type T given in its declaration.
- In the rule exIf for conditionals, the two branches must be well-typed in the context where the condition holds and does not hold, respectively. The additional assumption about the condition holding or not is realized by adding an axiom to the context.
- In order for a record update to be be well-typed, the record types of the operands must agree on the types of their common fields. The resulting record type consists of the union of the components comprising the two record types.
- A choice takes as argument an expression e that denotes a function from the quotiented type T to some other type T'. The function must be a congruence with respect to the predicate of the associated quotient type. The result is a well-typed expression that denotes a function from the quotient type T/q to T'. The condition that v and v' are distinct is important: without it, the congruence requirement would effectively disappear (because the corresponding formula would be trivially provable).

- For a case expression to be well-typed, the target expression e and all its patterns must have a common type T (the notion of well-typed pattern is defined later). In addition, e must match at least one of the patterns, as expressed by the disjunction quantified over i (which can be readily expanded into nested binary disjunctions). The order of the patterns in a case expression is relevant: the meaning is that every branch i assumes not only that e matches pattern p_i, but also that it does not match any of the other patterns p_j with $j < i$. The assumption that e matches p_i is introduced as a "positive" context cx_i^+, which also introduces the variables bound by the pattern p_i. The assumption that e does not match any of the previous patterns is introduced as a "negative" context cx_i^-, with a conjunction quantified over $j < i$ (which can be readily expanded into nested binary conjunctions; note that the empty conjunction, for $i = 1$, is regarded as true). The negative and positive contexts cx_i^- and cx_i^+ are added to the context cx (their relative order is actually unimportant) to assign some type T' to the branch expression e_i, which is also the type of the whole case expression. The rule exCase is analogous to the rule exIf for conditionals, with the extra complication that branches may bind variables and that their order matters.

- In order for a recursive let to be well-typed, it is necessary that the solution to the (in general, recursive) associated equation is unique, similarly to the requirement for op definitions in well-formed contexts. Indeed, as already mentioned, recursive let's capture "let def" as defined in [1], so it not surprising that they must satisfy requirements similar to op definitions.

- Application, abstractions, equalities, records, projections, embedders, relaxators, and quotienters are straightforward.

- Rules exSuper and exSub link the notion of well-typed expressions to the notion of subtyping. If an expression e has a subtype T, it also has any supertype T'. If an expression e has a supertype T', it also has any subtype T such that the associated predicate holds on e. Note that rule exSuper, in conjunction with rule stRefl, can be used to show that if an expression e has type T, it also has any type T' equivalent to T.

- The last three rules amount to treating expressions up to alpha equivalence, allowing to rename bound variables maintaining well-typedness. Without these rules, the Metaslang logic would be non-monotone, because extending a context with variable declarations may invalidate conclusions about the well-typedness of expressions that bind those variables (e.g. if $cx \vdash \lambda v\!:\!T.e : T'$ were provable then $cx, \mathsf{var}\ v\!:\!T \vdash \lambda v\!:\!T.e : T'$ would not be provable). These rules also allow variable hiding as described in [1].

3.7 Well-typed patterns

We define a ternary relation $\_ \vdash \_ : \_ \subseteq \mathit{Cx} \times \mathit{Pat} \times \mathit{Type}$ to capture well-typed patterns as

$$\frac{\begin{array}{c} cx \vdash T : \textsc{type} \\ v \notin \mathcal{V}(cx) \end{array}}{cx \vdash v\!:\!T : T} \quad (\textsc{paVar})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ cx \vdash p : T_j \end{array}}{cx \vdash \mathsf{emb}\ c_j\ p : \sum_i c_i\ T_i} \quad (\textsc{paEmbed})$$

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \textsc{type} \\ \forall i.\ \ cx \vdash p_i : T_i \\ \forall i,j.\ \ i \neq j \ \Rightarrow\ \mathcal{V}(p_i) \cap \mathcal{V}(p_j) = \emptyset \end{array}}{cx \vdash \{f_i \leftarrow p_i\}_i : \prod_i f_i\ T_i} \quad (\textsc{paRec})$$

$$\frac{\begin{array}{c} cx \vdash p : T \\ v \notin \mathcal{V}(cx) \cup \mathcal{V}(p) \end{array}}{cx \vdash (v\!:\!T \ \mathsf{as}\ p) : T} \quad (\textsc{paAs})$$

$$\frac{\begin{array}{c} cx \vdash p : T \\ cx \vdash T \prec_r T' \end{array}}{cx \vdash p : T'} \quad (\textsc{paSuper})$$

Explanation:

- A variable pattern can be introduced only if v is not already declared in the context. The reason is that, as shown in rule exCase for well-typed case expressions, the variables bound by a pattern are added to the context for the branch expression, and that context must be well-formed.
- Alias patterns also require the variable not to be already declared in the context.
- Embedding and record patterns are straightforward.
- The last rule links the notion of well-typed patterns to the notion of subtyping: if a pattern p has a subtype T, it also has any supertype T'. This is analogous to rule exSuper for expressions; however, for patterns, there is no rule corresponding to exSub, because a pattern introduces new variables, so no assumption on them is available to prove that a pattern of a supertype satisfies the predicate associated to the subtype. Note that rule paSuper, in conjunction with rule stRefl, can be used to show that if an expression e has type T, it also has any type T' equivalent to T.

3.8 Theorems

We define a binary relation $\_ \vdash \_ \subseteq \mathit{Cx} \times \mathit{Exp}$ to capture theorems as

$$\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ \mathsf{ax}\ [\overline{\beta}]\ e \in cx \\ \forall i.\ \ cx \vdash T_i : \textsc{type} \end{array}}{cx \vdash e[\overline{\beta}/\overline{T}]} \quad (\textsc{thAx})$$

$$\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ \mathsf{def}\ [\overline{\beta}]\ o = e \in cx \\ \forall i.\ \ cx \vdash T_i : \textsc{type} \end{array}}{cx \vdash o[\overline{T}] \equiv e[\overline{\beta}/\overline{T}]} \quad (\textsc{thDef})$$

$$\frac{\begin{array}{c} cx \vdash e \\ cx \vdash e_1 \equiv e_2 \\ e[e_1/e_2@\omega] \rightsquigarrow e' \\ \mathit{OKsbs}(e, e_1, e_2, \omega) \end{array}}{cx \vdash e'} \quad (\textsc{thSubst})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 : T \\ cx, \mathsf{ax}\ e_0 \vdash e_1 \equiv e_1' \\ cx, \mathsf{ax}\ \neg\ e_0 \vdash e_2 \equiv e_2' \end{array}}{cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 \equiv \mathsf{if}\ e_0\ e_1'\ e_2'} \quad (\textsc{thSubstIf})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i : T \\ \forall i.\ \ cx_i^- = \mathsf{ax}\ \bigwedge_{j<i} \forall pbnd(p_j).\ \neg\ pasm(p_j, e) \\ \forall i.\ \ cx_i^+ = \mathsf{var}\ pbnd(p_i), \mathsf{ax}\ pasm(p_i, e) \\ \forall i.\ \ cx, cx_i^-, cx_i^+ \vdash e_i \equiv e_i' \end{array}}{cx \vdash \mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i \equiv \mathsf{case}\ e\ \{p_i \rightarrow e_i'\}_i} \quad (\textsc{thSubstCase})$$

$$\frac{\begin{array}{c} cx \vdash e \\ cx \vdash T_1 \approx T_2 \\ e[T_1/T_2@\omega] \rightsquigarrow e' \end{array}}{cx \vdash e'} \quad (\textsc{thTySubst})$$

$$\frac{\begin{array}{c} cx \vdash e : \mathsf{Bool} \rightarrow \mathsf{Bool} \\ v \notin \mathcal{FV}(e) \end{array}}{cx \vdash e\ \mathsf{true} \wedge e\ \mathsf{false} \Leftrightarrow (\forall v{:}\mathsf{Bool}.\ e\ v)} \quad (\textsc{thBool})$$

$$\frac{\begin{array}{c} cx \vdash e_1 : T \\ cx \vdash e_2 : T \\ cx \vdash e : T \rightarrow T' \end{array}}{cx \vdash e_1 \equiv e_2 \Rightarrow e\ e_1 \equiv e\ e_2} \quad (\textsc{thCongr})$$

$$\frac{\begin{array}{c} cx \vdash e_1 : T \rightarrow T' \\ cx \vdash e_2 : T \rightarrow T' \\ v \notin \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \end{array}}{cx \vdash e_1 \equiv e_2 \Leftrightarrow (\forall v{:}T.\ e_1\ v \equiv e_2\ v)} \quad (\textsc{thExt})$$

$$\frac{\begin{array}{c} cx \vdash (\lambda v{:}T.e)\ e' : T' \\ OKsbs(e, v, e') \end{array}}{cx \vdash (\lambda v{:}T.e)\ e' \equiv e[v/e']} \quad (\textsc{thAbs})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 : T \\ cx, \mathsf{ax}\ e_0 \vdash e_1 \equiv e' \\ cx, \mathsf{ax}\ \neg\ e_0 \vdash e_2 \equiv e' \end{array}}{cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 \equiv e'} \quad (\textsc{thIf})$$

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \textsc{type} \\ v, \overline{v} \text{ all distinct} \end{array}}{cx \vdash \forall v{:}\prod_i f_i\ T_i.\ (\exists \overline{v}{:}\overline{T}.\ v \equiv \{f_i \leftarrow v_i\}_i)} \quad (\textsc{thRec})$$

$$\frac{cx \vdash \{f_i \leftarrow e_i\}_i : \prod_i f_i\ T_i}{cx \vdash \{f_i \leftarrow e_i\}_i.f_j \equiv e_j} \quad (\textsc{thProj})$$

$$\frac{\begin{array}{c} cx \vdash \{f_i \leftarrow e_i\}_i : \prod_i f_i\ T_i \\ cx \vdash \{f_j' \leftarrow e_j'\}_j : \prod_j f_j'\ T_j' \\ f_k \notin \overline{f}' \end{array}}{cx \vdash (\{f_i \leftarrow e_i\}_i \ll \{f_j' \leftarrow e_j'\}_j).f_k \equiv e_k} \quad (\textsc{thRecUpd1})$$

$$\frac{\begin{array}{c} cx \vdash \{f_i \leftarrow e_i\}_i : \prod_i f_i\ T_i \\ cx \vdash \{f'_j \leftarrow e'_j\}_j : \prod_j f'_j\ T'_j \end{array}}{cx \vdash (\{f_i \leftarrow e_i\}_i \ll \{f'_j \leftarrow e'_j\}_j).f'_k \equiv e'_k} \quad (\textsc{thRecUpd2})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ v \neq v' \end{array}}{cx \vdash \forall v : \sum_i c_i\ T_i.\ \bigvee_i \exists v' : T_i.\ v \equiv \mathsf{emb}\ c_i\ v'} \quad (\textsc{thEmbSurj})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ j \neq k \\ v \neq v' \end{array}}{cx \vdash \forall v : T_j, v' : T_k.\ \mathsf{emb}\ c_j\ v \not\equiv \mathsf{emb}\ c_k\ v'} \quad (\textsc{thEmbDist})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \textsc{type} \\ v \neq v' \end{array}}{cx \vdash \forall v : T_j, v' : T_j.\ v \not\equiv v' \Rightarrow \mathsf{emb}\ c_j\ v \not\equiv \mathsf{emb}\ c_j\ v'} \quad (\textsc{thEmbInj})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ v \neq v' \end{array}}{cx \vdash \forall v : T/q.\ (\exists v' : T.\ \mathsf{quo}_q\ v' \equiv v)} \quad (\textsc{thQuotSurj})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ v \neq v' \end{array}}{cx \vdash \forall v : T, v' : T.\ q\ \langle v, v' \rangle \Leftrightarrow \mathsf{quo}_q\ v \equiv \mathsf{quo}_q\ v'} \quad (\textsc{thQuotEqCls})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{ch}_q\ e : T/q \rightarrow T' \\ v \notin \mathcal{FV}(e) \end{array}}{cx \vdash \forall v : T.\ (\mathsf{ch}_q\ e)\ (\mathsf{quo}_q\ v) \equiv e\ v} \quad (\textsc{thChoose})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i : T \\ \forall i.\quad cx_i^- = \mathsf{ax}\ \bigwedge_{j<i} \forall pbnd(p_j).\ \neg\ pasm(p_j, e) \\ \forall i.\quad cx_i^+ = \mathsf{var}\ pbnd(p_i), \mathsf{ax}\ pasm(p_i, e) \\ \forall i.\quad cx, cx_i^-, cx_i^+ \vdash e_i \equiv e' \\ \forall i.\quad \mathcal{FV}(e') \cap \mathcal{V}(p_i) = \emptyset \end{array}}{cx \vdash \mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i \equiv e'} \quad (\textsc{thCase})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{case}\ e\ \{\ldots p_i \rightarrow e_i \ldots\} : T \\ v \in \mathcal{V}(p_i) \\ v' \notin \mathcal{V}(p_i) \cup \mathcal{FV}(e_i) \cup \mathcal{CV}(e_i, v) \end{array}}{cx \vdash \mathsf{case}\ e\ \{\ldots p_i \rightarrow e_i \ldots\} \equiv \mathsf{case}\ e\ \{\ldots p_i[v/v'] \rightarrow e_i[v/v'] \ldots\}} \quad (\textsc{thCaseAlpha})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{letr}\ \{v_i : T_i \leftarrow e_i\}_i\ e : T \\ cx, \mathsf{var}\ \overline{v} : \overline{T}, \mathsf{ax}\ \bigwedge_i v_i \equiv e_i \vdash e \equiv e' \\ \overline{v} \cap \mathcal{FV}(e') = \emptyset \end{array}}{cx \vdash \mathsf{letr}\ \{v_i : T_i \leftarrow e_i\}_i\ e \equiv e'} \quad (\textsc{thLetRec})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e : T \\ v'_j \notin \overline{v} \cup \mathcal{FV}(e) \cup \mathcal{CV}(e, v_j) \cup \bigcup_i (\mathcal{FV}(e_i) \cup \mathcal{CV}(e_i, v_j)) \\ \forall i \neq j.\ \ v'_i = v_i \end{array}}{cx \vdash \mathsf{letr}\ \{v_i\!:\!T_i \!\leftarrow\! e_i\}_i\ e \equiv \mathsf{letr}\ \{v'_i\!:\!T_i \!\leftarrow\! e_i[v_j/v'_j]\}_i\ e[v_j/v'_j]} \quad (\textsc{thLetRecAlpha})$$

Explanation:

- Axioms in a well-formed context are readily instantiated into theorems, via rule thAx. More precisely, the type variables over which the axiom is polymorphic are replaced with well-formed types.
- Similarly, op definitions in a well-formed context are readily instantiated into theorems, via rule thDef.
- Rule thSubst allows the occurrence of an expression e_1 in a theorem e to be replaced with an expression e_2 provably equal to e_1. Rules thSubstIf and thSubstCase are variants that take into account the context of conditional and case branches.
- Rule thTySubst allows the occurrence of a type T_1 in a theorem e to be replaced with an equivalent type T_2.
- Rule thBool asserts that true and false are the only values of type Bool. Note that the variable v used in the universal quantification must not occur free in e, otherwise it would be captured.
- Rule thCongr asserts that equality is a congruence with respect to any function.
- Rule thExt says that a function is characterized by its values over all the values of its domain, i.e. by extensionality.
- Rule thAbs defines the semantics of lambda abstraction: the bound variable v is replaced with the argument e' in the body e. The premise of the rule says that the application is well-typed.
- Rule thIf defines the semantics of conditionals: a conditional equals an expression if both branches do, in the contexts extended with the assumption that the condition is true and false, respectively. Note the premise that requires the conditional to be well-typed (but the type T is not used in the rest of the rule).
- Rules thRec and thProj characterize record types. The first rule says that every value of a record type is a record; note that the variables v and $\overline{v}$ used in the quantifiers of the theorem must be all distinct and not already declared in the context. The second rule defines the semantics of projections, at the same time constraining record construction, viewed as a function, to be injective, because if records with different arguments were mapped to the same value of the record type, the theorem asserted by the rule would be violated.
- Rules thRecUpd1 and thRecUpd2 define the semantics of record updates: fields in either the first or second operand retain their original values, while for common fields the second operand takes precedence.
- Rules thEmbSurj, thEmbDist, and thEmbInj characterize sum types. They say that every value of a sum type is the image of some constructor (i.e. the constructors are collectively surjective), that the images of distinct constructors are disjoint, and that each constructor is injective.
- Rules thQuotSurj and thQuotEqCls characterize quotient types. The first rules says that quo_q is a surjective function (i.e. every quotient value is obtained by applying it to some value of the quotiented type). The second rule says that quo_q maps each value of the quotiented type to its equivalence class, which is a value of the quotient type.

- Rule THCHOOSE defines the semantics of choices: the result of applying $\mathsf{ch}_q\ e$ to an equivalence class of the quotient type is the same as applying the choice argument e to any member of the equivalence class. Recall that the well-typedness of $\mathsf{ch}_q\ e$ includes the fact that e maps equivalent values to the same value (cf. rule EXCHOOSE).
- Rule THCASE defines the semantics of case expressions. The context for each branch is extended in the same way as in rule EXCASE (which defines the well-typedness of case expressions). Instead of requiring every branch to be well-typed, rule THCASE requires the expression in every branch to be provably equal to some expression e'. If such an expression has no free variables bound by the patterns, then the case expression is provably equal to e'. The requirement about no free variables bound by the patterns ensures that the resulting expression is well-typed in the unextended context in which the case expression is well-typed. This rule is analogous to THIF for conditionals, with the extra complication that branches may bind variables and that their order matters.
- Rule THCASEALPHA says that alpha-equivalent case expressions are equal in the logic. This rule is needed to manipulate case expressions that bind variables that happen to be in the context: since rule THCASE extends the context with the variables bound by the patterns, rule THCASEALPHA must be first used to rename the pattern variables that happen to be declared in the non-extended context; otherwise, the extended context would not be well-formed.
- Rule THLETREC defines the semantics of recursive let's. The context is extended with the equality derived by the bindings and if the body e is provably equal to an expression e' where the let-bound variables $\overline{v}$ do not occur free, the whole recursive let is provably equal to e'. The requirement that $\overline{v}$ do not occur free in e' ensures that e' is well-typed in the unextended context in which the recursive let is well-typed.
- The purpose of rule THLETRECALPHA is analogous to THCASEALPHA.

3.9 Proofs

The previous subsections have defined judgements of the forms

$$\begin{array}{l}\vdash cx : \textsc{context}\\ \vdash sp : \textsc{spec}\\ cx \vdash T : \textsc{type}\\ cx \vdash T_1 \approx T_2\\ cx \vdash T_1 \prec_r T_2\\ cx \vdash e : T\\ cx \vdash p : T\\ cx \vdash e\end{array}$$

by means of a set of inductive rules.

A proof of a judgement is a finite sequence of judgements that ends with the proved judgement and where each judgement in the sequence is derived from preceding judgements using some rule.

[[[TO DO: Make sure that the rules for theorems are "sufficient", i.e. all truths "of interest" are indeed theorems derivable from the rules. Even though higher-order logic is notoriously incomplete, in practice theorem provers like PVS and HOL are sufficient to prove desired properties of formalized concepts without running into theoretical limitations. Perhaps the requirement boils down to prove completeness with respect to so-called "general models" (cf. [2]).]]]

4 Properties

[[[TO DO]]]

This section proves certain (meta-)properties of the proof theory introduced in §3. For instance, it proves that if the judgement $cx \vdash e : T$ can be derived then also $cx \vdash T : \textsc{type}$ can (i.e. the type of a well-typed expression is well-formed). These properties serve to validate the proof theory, i.e. to increase confidence that the proof theory correctly captures our intentions and requirements.

5 Models

[[[TO DO]]]

This section defines the notion of model of a context (recall that specs are contexts without variable and type variable declarations).

A model of a context is a mapping from names declared in the context to suitable set-theoretic entities. For instance, a type name τ of arity n is mapped to an n-ary function over sets (if $n = 0$, the model maps the type name simply to a set). The mapping is extended to all well-formed types, which are mapped to sets, and to all well-typed expressions, which are mapped to elements of the sets that their types map to. The model must satisfy all the type definitions, op definitions, and axioms of the context.

It should be possible to prove the soundness of the rules to derive assertions with respect to models.

Since higher-order logic is notoriously incomplete, it is not possible to prove completeness of the rules to derive assertions. However, it should be possible to prove completeness with respect to general (a.k.a. Henkin) models. A general model is one in which the type $T_1 \rightarrow T_2$ is a subset of all functions from T_1 to T_2, and not necessarily the set of all such functions (as in standard models). Since there are more general models than standard models (a standard model is also a general model but not all general models are standard models), fewer formulas are true in all general models than in all standard models.

Perhaps this section should also contain a proof of the consistency of the Metaslang logic, analogously to the proof of the consistency of the higher-order logic defined in [2].

References

[1] Kestrel Institute and Kestrel Technology LLC. *Specware 4.1 Language Manual.* Available at www.specware.org.

[2] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Thruth Through Proof.* Academic Press, 1986.

[3] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report CSL–97–2R, SRI International, August 1997. Revised March 1999.

[4] *The HOL System Description*, July 1997.