The Logic of Metaslang

Alessandro Coglio

July 5, 2004

1 Introduction

This document formally defines the logic of the Metaslang language [1]. It does not define all of Metaslang; for example, concrete syntax and type inference are not defined. The abstract syntax defined in this document is meant to capture Metaslang specs after Specware has fully processed them and checked them for validity; for example, it assumes that overloaded symbols have been disambiguated and missing types have been inferred. It is possible to write Metaslang specs that correspond very closely to the syntax defined in this document, by tediously writing all types etc.

This document draws 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 (ordered) 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, $A \xrightarrow{\mathrm{p}} B$ 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. $\{f \in A \xrightarrow{\mathrm{p}} B \mid \forall a \in A.\ \exists b \in B.\ \langle a, b\rangle \in f\}$; 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 $f : A \xrightarrow{\mathrm{p}} B$, $f : A \to B$, and $f : A \hookrightarrow B$ for $f \in A \xrightarrow{\mathrm{p}} B$, $f \in A \to B$, 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 $i \neq j$.

2.2 Types

We inductively define the set of types as

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

where *Exp* is defined later.[1]

Explanation:

- There is a type Bool for boolean (i.e. truth) values.
- A name α is a type variable.

[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.

- 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 $T_1 \rightarrow T_2$ is obtained by combining a domain type T_1 and a range type T_2.
- Product types $\prod_i f_i\ T_i$ (resp. sum types $\sum_i c_i\ T_i$) include explicit fields f_i (resp. constructors c_i). Note that product types can be empty (denoted $\prod \epsilon$), while sum types cannot. All the fields (resp. constructors) of a product (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 product type where a constructor has no type, and add the empty tuple 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 abbreviate $\prod_{\pi_i} T_i$ to $\prod T_i$. Thus, product types as defined above capture both record and product types as defined in [1].

2.3 Expressions

We inductively define the set of expressions as

$$
\begin{aligned}
\mathit{Exp} = {} & \{v \mid v \in \mathcal{N}\} \\
& + \{o[\overline{T}] \mid o \in \mathcal{N} \ \wedge\ \overline{T} \in \mathit{Type}^*\} \\
& + \{e_1\ e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\lambda v{:}T.e \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type} \ \wedge\ e \in \mathit{Exp}\} \\
& + \{e_1 \equiv e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\mathsf{if}\ e_0\ e_1\ e_2 \mid e_0, e_1, e_2 \in \mathit{Exp}\} \\
& + \{\{f_1 \leftarrow e_1 \ldots f_n \leftarrow e_n\} \mid \overline{f} \in \mathcal{N}^{(*)} \ \wedge\ \overline{e} \in \mathit{Exp}^*\} \\
& + \{e.f \mid e \in \mathit{Exp} \ \wedge\ f \in \mathcal{N}\} \\
& + \{\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j \mid \textstyle\sum_i c_i\ T_i \in \mathit{Type}\} \\
& + \{\mathsf{rel}_r \mid r \in \mathit{Exp}\} \\
& + \{\mathsf{res}_r\ e \mid r, e \in \mathit{Exp}\} \\
& + \{\mathsf{quo}_q \mid q \in \mathit{Exp}\} \\
& + \{\mathsf{ch}_q\ e \mid q, e \in \mathit{Exp}\} \\
& + \{\mathsf{case}\ e\ \{p_1 \rightarrow e_1 \ldots p_n \rightarrow e_n\} \mid e \in \mathit{Exp} \ \wedge\ \overline{p} \in \mathit{Pat}^+ \ \wedge\ \overline{e} \in \mathit{Exp}^+\} \\
& + \{\mathsf{letr}\ v{:}T \leftarrow e\ \mathsf{in}\ e' \mid v \in \mathcal{N} \ \wedge\ T \in \mathit{Type} \ \wedge\ e, e' \in \mathit{Exp}\}
\end{aligned}
$$

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 tuple $\{f_i \leftarrow e_i\}_i$ consists of a sequence of pairs that associate expressions to distinct field names.
- 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 relaxator rel_r (resp. quotienter quo_q) is decorated by the associated predicate, which defines the subtype (resp. quotient type).
- A restriction $\mathsf{res}_r\ e$ (resp. choice $\mathsf{ch}_q\ e$) is also decorated by the associated predicate.
- A recursive let $\mathsf{letr}\ v{:}T \leftarrow e\ \mathsf{in}\ e'$ captures "let def" as defined in [1].
- An application $e_1\ e_2$, an equality $e_1 \equiv e_2$, a projection $e.f$, 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])

$$
\begin{array}{rcll}
\mathsf{true} & \longrightarrow & \lambda v\!:\!\mathsf{Bool}.v \equiv \lambda v\!:\!\mathsf{Bool}.v & [v \text{ fresh}] \\
\mathsf{false} & \longrightarrow & \lambda v\!:\!\mathsf{Bool}.v \equiv \lambda v\!:\!\mathsf{Bool}.\mathsf{true} & [v \text{ fresh}] \\
\neg\, e & \longrightarrow & \mathsf{if}\ e\ \mathsf{false}\ \mathsf{true} & \\
e_1 \wedge e_2 & \longrightarrow & \mathsf{if}\ e_1\ e_2\ \mathsf{false} & \\
e_1 \vee e_2 & \longrightarrow & \mathsf{if}\ e_1\ \mathsf{true}\ e_2 & \\
e_1 \Rightarrow e_2 & \longrightarrow & \mathsf{if}\ e_1\ e_2\ \mathsf{true} & \\
e_1 \Leftrightarrow e_2 & \longrightarrow & e_1 \equiv e_2 & [\text{booleans}] \\
e_1 \not\equiv e_2 & \longrightarrow & \neg\,(e_1 \equiv e_2) & \\
\forall v\!:\!T.\ e & \longrightarrow & \lambda v\!:\!T.e \equiv \lambda v\!:\!T.\mathsf{true} & \\
\exists v\!:\!T.\ e & \longrightarrow & \neg\,(\forall v\!:\!T.\ \neg\, e) & \\
\exists!\, v\!:\!T.\ e & \longrightarrow & \exists v\!:\!T.\ (e \wedge \forall v'\!:\!T.\ e[v'/v] \Rightarrow v \equiv v') & [v' \text{ fresh}] \\
\mathsf{let}\ p \leftarrow e\ \mathsf{in}\ e' & \longrightarrow & \mathsf{case}\ e\ \{p \rightarrow e'\} & \\
\langle \overline{e} \rangle & \longrightarrow & \{\pi_i \leftarrow e_i\}_i & \\
\forall v_1\!:\!T_1, \ldots, v_n\!:\!T_n.\ e & \longrightarrow & \forall v_1\!:\!T_1.\ \ldots\ \forall v_n\!:\!T_n.\ e & \\
\exists v_1\!:\!T_1, \ldots, v_n\!:\!T_n.\ e & \longrightarrow & \exists v_1\!:\!T_1.\ \ldots\ \exists v_n\!:\!T_n.\ e & \\
\forall \overline{v}\!:\!\overline{T}.\ e & \longrightarrow & \forall v_1\!:\!T_1, \ldots, v_n\!:\!T_n.\ e & [|\overline{v}| = |\overline{T}|] \\
\exists \overline{v}\!:\!\overline{T}.\ e & \longrightarrow & \exists v_1\!:\!T_1, \ldots, v_n\!:\!T_n.\ e & [|\overline{v}| = |\overline{T}|]
\end{array}
$$

where substitution $e[v'/v]$ is defined later. The adjective "fresh" means that the variable does not appear anywhere else (in particular, $v' \neq v$ in the $\exists!\, v : T.\ e$ abbreviation). We use the abbreviation $e_1 \Leftrightarrow e_2$ only for expressions of type Bool (defined later). Note that both $\forall \epsilon.\ e$ and $\exists \epsilon.\ e$ stand for 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, 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 $\mathcal{FV} : \mathit{Exp} \to \mathcal{P}_\omega(\mathcal{N})$ returns the free variables of an expression

$$\begin{array}{ll}
\mathcal{FV}(o[\overline{T}]) & = \emptyset \\
\mathcal{FV}(v) & = \{v\} \\
\mathcal{FV}(e_1\ e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\lambda v\!:\!T.e) & = \mathcal{FV}(e) - \{v\} \\
\mathcal{FV}(e_1 \equiv e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\mathsf{if}\ e_0\ e_1\ e_2) & = \mathcal{FV}(e_0) \cup \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\{f_i \leftarrow e_i\}_i) & = \bigcup_i \mathcal{FV}(e_i) \\
\mathcal{FV}(e.f) & = \mathcal{FV}(e) \\
\mathcal{FV}(\mathsf{emb}\ c_j) & = \emptyset \\
\mathcal{FV}(\mathsf{rel}_r) & = \mathcal{FV}(r) \\
\mathcal{FV}(\mathsf{res}_r\ e) & = \mathcal{FV}(r) \cup \mathcal{FV}(e) \\
\mathcal{FV}(\mathsf{quo}_q) & = \mathcal{FV}(q) \\
\mathcal{FV}(\mathsf{ch}_q\ e) & = \mathcal{FV}(q) \cup \mathcal{FV}(e) \\
\mathcal{FV}(\mathsf{case}\ e\ \{p_i \to e_i\}_i) & = \mathcal{FV}(e) \cup \bigcup_i(\mathcal{FV}(e_i) - \mathcal{V}(p_i)) \\
\mathcal{FV}(\mathsf{letr}\ v\!:\!T \leftarrow e\ \mathsf{in}\ e') & = (\mathcal{FV}(e) \cup \mathcal{FV}(e')) - \{v\}
\end{array}$$

where $\mathcal{V}$ on patterns is defined later.

2.4 Patterns

We inductively define the set of patterns as

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

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 tuple 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.

Relax and quotient patterns as defined in [1] are not modeled here due to their unclear purpose and/or semantics. Indeed, it is argued that they should be removed from [1].

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

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

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

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

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

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

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

$$\mathit{pasm}(p, e) = \begin{cases} e \equiv \mathit{p2e}(p) \wedge v \equiv e & \text{if} \quad p = v{:}T\ \mathsf{as}\ \ldots \\ e \equiv \mathit{p2e}(p) & \text{otherwise} \end{cases}$$

2.5 Contexts

We define the set of context elements as

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

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 $\mathsf{op}\ o : [\overline{\alpha}]\ T$ introduces an op name with an associated type, polymorphic in the explicit type variables.
- A type definition $\mathsf{def}\ \tau(\overline{\alpha}) = T$ 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 $\mathsf{def}\ [\overline{\alpha}]\ o = e$ 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 $\mathsf{ax}\ [\overline{\alpha}]\ e$ 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 $\mathsf{tvar}\ \alpha$ introduces a type variable.
- A variable declaration $\mathsf{var}\ v{:}T$ introduces a variable with a type.

We introduce the following abbreviations

$$\begin{array}{rcl} \mathsf{tvar}\ \alpha_1, \ldots, \alpha_n & \longrightarrow & \mathsf{tvar}\ \alpha_1, \ldots, \mathsf{tvar}\ \alpha_n \\ \mathsf{var}\ v_1{:}T_1, \ldots, v_n{:}T_n & \longrightarrow & \mathsf{var}\ v_1{:}T_1, \ldots, \mathsf{var}\ v_n{:}T_n \end{array}$$

We define the set of contexts as

$$Cx = CxElem^*$$

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

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

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

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

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

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

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

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

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

2.6 Specs

We define the set of spec(ification)s as

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

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

2.7 Substitutions

Given type variables $\overline{\alpha}$ and types $\overline{S}$ with $|\overline{\alpha}| = |\overline{S}|$, we define the substitution of $\overline{\alpha}$ with $\overline{S}$ in types, type sequences, expressions, and patterns as

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

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

$$\begin{aligned}
o[\overline{T}][\overline{S}/\overline{\alpha}] &= o[\overline{T}[\overline{S}/\overline{\alpha}]] \\
v[\overline{S}/\overline{\alpha}] &= v \\
(e_1\ e_2)[\overline{S}/\overline{\alpha}] &= e_1[\overline{S}/\overline{\alpha}]\ e_2[\overline{S}/\overline{\alpha}] \\
(\lambda v{:}T.e)[\overline{S}/\overline{\alpha}] &= \lambda v{:}T[\overline{S}/\overline{\alpha}].e[\overline{S}/\overline{\alpha}] \\
(e_1 \equiv e_2)[\overline{S}/\overline{\alpha}] &= e_1[\overline{S}/\overline{\alpha}] \equiv e_2[\overline{S}/\overline{\alpha}] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[\overline{S}/\overline{\alpha}] &= \mathsf{if}\ e_0[\overline{S}/\overline{\alpha}]\ e_1[\overline{S}/\overline{\alpha}]\ e_2[\overline{S}/\overline{\alpha}] \\
\{f_i \leftarrow e_i\}_i[\overline{S}/\overline{\alpha}] &= \{f_i \leftarrow e_i[\overline{S}/\overline{\alpha}]\}_i \\
(e.f)[\overline{S}/\overline{\alpha}] &= e[\overline{S}/\overline{\alpha}].f \\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\right)[\overline{S}/\overline{\alpha}] &= \mathsf{emb}_{(\sum_i c_i\ T_i)[\overline{S}/\overline{\alpha}]}\ c_j \\
\mathsf{rel}_r[\overline{S}/\overline{\alpha}] &= \mathsf{rel}_{r[\overline{S}/\overline{\alpha}]} \\
(\mathsf{res}_r\ e)[\overline{S}/\overline{\alpha}] &= \mathsf{res}_{r[\overline{S}/\overline{\alpha}]}\ e[\overline{S}/\overline{\alpha}] \\
\mathsf{quo}_q[\overline{S}/\overline{\alpha}] &= \mathsf{quo}_{q[\overline{S}/\overline{\alpha}]} \\
(\mathsf{ch}_q\ e)[\overline{S}/\overline{\alpha}] &= \mathsf{ch}_{q[\overline{S}/\overline{\alpha}]}\ e[\overline{S}/\overline{\alpha}] \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[\overline{S}/\overline{\alpha}] &= \mathsf{case}\ e[\overline{S}/\overline{\alpha}]\ \{p_i[\overline{S}/\overline{\alpha}] \rightarrow e_i[\overline{S}/\overline{\alpha}]\}_i \\
(\mathsf{letr}\ v{:}T \leftarrow e\ \mathsf{in}\ e')[\overline{S}/\overline{\alpha}] &= \mathsf{letr}\ v{:}T[\overline{S}/\overline{\alpha}] \leftarrow e[\overline{S}/\overline{\alpha}]\ \mathsf{in}\ e'[\overline{S}/\overline{\alpha}]
\end{aligned}$$

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

Given variables $\overline{v}$ and expressions $\overline{d}$ with $|\overline{v}| = |\overline{d}|$, we define the substitution of $\overline{v}$ with $\overline{d}$ in expressions as

$$\begin{aligned}
o[\overline{T}][\overline{d}/\overline{v}] &= o[\overline{T}] \\
v[\overline{d}/\overline{v}] &= \begin{cases} d_i & \text{if} \quad v = v_i \\ v & \text{otherwise} \end{cases} \\
(e_1\ e_2)[\overline{d}/\overline{v}] &= e_1[\overline{d}/\overline{v}]\ e_2[\overline{d}/\overline{v}] \\
(\lambda v\!:\!T.e)[\overline{d}/\overline{v}] &= \lambda v\!:\!T.e[\overline{d}/\overline{v}] \\
(e_1 \equiv e_2)[\overline{d}/\overline{v}] &= e_1[\overline{d}/\overline{v}] \equiv e_2[\overline{d}/\overline{v}] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[\overline{d}/\overline{v}] &= \mathsf{if}\ e_0[\overline{d}/\overline{v}]\ e_1[\overline{d}/\overline{v}]\ e_2[\overline{d}/\overline{v}] \\
\{f_i \leftarrow e_i\}_i[\overline{d}/\overline{v}] &= \{f_i \leftarrow e_i[\overline{d}/\overline{v}]\}_i \\
(e.f)[\overline{d}/\overline{v}] &= e[\overline{d}/\overline{v}].f \\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\right)[\overline{d}/\overline{v}] &= \mathsf{emb}_{\sum_i c_i\ T_i}\ c_j \\
\mathsf{rel}_r[\overline{d}/\overline{v}] &= \mathsf{rel}_{r[\overline{d}/\overline{v}]} \\
(\mathsf{res}_r\ e)[\overline{d}/\overline{v}] &= \mathsf{res}_{r[\overline{d}/\overline{v}]}\ e[\overline{d}/\overline{v}] \\
\mathsf{quo}_q[\overline{d}/\overline{v}] &= \mathsf{quo}_{q[\overline{d}/\overline{v}]} \\
(\mathsf{ch}_q\ e)[\overline{d}/\overline{v}] &= \mathsf{ch}_{q[\overline{d}/\overline{v}]}\ e[\overline{d}/\overline{v}] \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[\overline{d}/\overline{v}] &= \mathsf{case}\ e[\overline{d}/\overline{v}]\ \{p_i \rightarrow e_i[\overline{d}/\overline{v}]\}_i \\
(\mathsf{letr}\ v\!:\!T \leftarrow e\ \mathsf{in}\ e')[\overline{d}/\overline{v}] &= \mathsf{letr}\ v\!:\!T \leftarrow e[\overline{d}/\overline{v}]\ \mathsf{in}\ e'[\overline{d}/\overline{v}]
\end{aligned}$$

The occurrence of a subexpression in a superexpression can be identified by a sequence of natural numbers that describe a path through the tree. We define the set of all possible occurrences as

$$Occ = \mathbf{N}^*$$

The substitution of an expression d with d' occurring at $\omega \in Occ$ in an expression e, denoted by $e[d'/d\,@\,\omega]$, is defined as

$$\begin{aligned}
d[d'/d\,@\,\epsilon] &= d' \\
(e_1\ e_2)[d'/d\,@\,1,\omega] &= e_1[d'/d\,@\,\omega]\ e_2 \\
(e_1\ e_2)[d'/d\,@\,2,\omega] &= e_1\ e_2[d'/d\,@\,\omega] \\
(\lambda v\!:\!T.e)[d'/d\,@\,0,\omega] &= \lambda v\!:\!T.e[d'/d\,@\,\omega] \\
(e_1 \equiv e_2)[d'/d\,@\,1,\omega] &= e_1[d'/d\,@\,\omega] \equiv e_2 \\
(e_1 \equiv e_2)[d'/d\,@\,2,\omega] &= e_1 \equiv e_2[d'/d\,@\,\omega] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d'/d\,@\,0,\omega] &= \mathsf{if}\ e_0[d'/d\,@\,\omega]\ e_1\ e_2 \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d'/d\,@\,1,\omega] &= \mathsf{if}\ e_0\ e_1[d'/d\,@\,\omega]\ e_2 \\
(\mathsf{if}\ e_0\ e_1\ e_2)[d'/d\,@\,2,\omega] &= \mathsf{if}\ e_0\ e_1\ e_2[d'/d\,@\,\omega] \\
\{f_i \leftarrow e_i\}_i[d'/d\,@\,i,\omega] &= \{f_1 \leftarrow e_1 \ldots f_i \leftarrow e_i[d'/d\,@\,\omega] \ldots f_n \leftarrow e_n\} \\
(e.f)[d'/d\,@\,0,\omega] &= e[d'/d\,@\,\omega].f \\
(\mathsf{res}_r\ e)[d'/d\,@\,0,\omega] &= \mathsf{res}_r\ e[d'/d\,@\,\omega] \\
(\mathsf{ch}_q\ e)[d'/d\,@\,0,\omega] &= \mathsf{ch}_q\ e[d'/d\,@\,\omega] \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[d'/d\,@\,0,\omega] &= \mathsf{case}\ e[d'/d\,@\,\omega]\ \{p_i \rightarrow e_i\}_i \\
(\mathsf{case}\ e\ \{p_i \rightarrow e_i\}_i)[d'/d\,@\,i,\omega] &= \mathsf{case}\ e\ \{p_1 \rightarrow e_1 \ldots p_i \rightarrow e_i[d'/d\,@\,\omega] \ldots p_n \rightarrow e_n\} \\
(\mathsf{letr}\ v\!:\!T \leftarrow e\ \mathsf{in}\ e')[d'/d\,@\,0,\omega] &= \mathsf{letr}\ v\!:\!T \leftarrow e[d'/d\,@\,\omega]\ \mathsf{in}\ e' \\
(\mathsf{letr}\ v\!:\!T \leftarrow e\ \mathsf{in}\ e')[d'/d\,@\,1,\omega] &= \mathsf{letr}\ v\!:\!T \leftarrow e\ \mathsf{in}\ e'[d'/d\,@\,\omega]
\end{aligned}$$

Note that this substitution operation is partial, i.e. it is not defined for all e, d, d', and ω. Note also that it is not allowed to do substitutions in the subtype and quotient type predicates that decorate relaxators, quotienters, restrictions, and choices.

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 assertions. The rules to derive such assertions 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

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

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

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ o \in \mathcal{N} - \mathcal{ON}(cx) \\ \overline{\alpha} \in \mathcal{N}^{(*)} \\ cx, \mathsf{tvar}\ \overline{\alpha} \vdash T : \textsc{type}\end{array}}{\vdash cx, \mathsf{op}\ o : [\overline{\alpha}]\ T : \textsc{context}} \quad (\textsc{cxOdec})$$

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

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

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \overline{\alpha} \in \mathcal{N}^{(*)} \\ cx, \mathsf{tvar}\ \overline{\alpha} \vdash e : \mathsf{Bool}\end{array}}{\vdash cx, \mathsf{ax}\ [\overline{\alpha}]\ e : \textsc{context}}\quad (\textsc{cxAx})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \alpha \in \mathcal{N} - \mathcal{TV}(cx)\end{array}}{\vdash cx, \mathsf{tvar}\ \alpha : \textsc{context}}\quad (\textsc{cxTVdec})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ v \in \mathcal{N} - \mathcal{V}(cx) \\ cx \vdash T : \textsc{type}\end{array}}{\vdash cx, \mathsf{var}\ v\!:\!T : \textsc{context}}\quad (\textsc{cxVdec})$$

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 $\mathsf{op}\ o : [\overline{\alpha}]\ T$ 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 $\overline{\alpha}$, which must be distinct. Note that we do not require that all type variables in T are among $\overline{\alpha}$, 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 $\mathsf{def}\ \tau(\overline{\alpha}) = T$ 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 $\overline{\alpha}$, 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 $\overline{\alpha}$, 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 $\overline{\alpha}'$ from the type variables $\overline{\alpha}$ 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 $T[\overline{\alpha}'/\overline{\alpha}]$. Of course, it is possible that $\overline{\alpha}' = \overline{\alpha}$. 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 $e[o[\overline{\alpha}']/v]$. The uniqueness of the solution is expressed as a theorem (defined later) in cx extended with the type variables $\overline{\alpha}'$.

- 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 $\overline{\alpha}$. As in other cases, all type variables in e are automatically in $\overline{\alpha}$ in well-formed specs, but that is not required in well-formed contexts in general.

- A type variable declaration $\mathsf{tvar}\ \alpha$ can be added to cx if α 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 types

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

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

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

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

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

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

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

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

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

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, product, 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 tyProd because a product type can have zero factors (unlike a sum type that always has at least one summand).

- For subtypes, we require the predicate to have type $T \to \mathsf{Bool}$, 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 $T \times T \to \mathsf{Bool}$, etc. (as proved later).

3.3 Type equivalence

We define a ternary relation $\_ \vdash \_ \sim \_ \subseteq Cx \times Type \times Type$ to capture type equivalence as

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

$$\frac{cx \vdash T : \textsc{type}}{cx \vdash T \sim T}\quad(\textsc{TeRefl})$$

$$\frac{cx \vdash T_1 \sim T_2}{cx \vdash T_2 \sim T_1}\quad(\textsc{TeSymm})$$

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

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \mathsf{ty}\ \tau : n \in cx \\ \forall i.\ \ cx \vdash T_i \sim T'_i\end{array}}{cx \vdash \tau(\overline{T}) \sim \tau(\overline{T}')}\quad(\textsc{TeInst})$$

$$\frac{\begin{array}{c}cx \vdash T_1 \sim T'_1 \\ cx \vdash T_2 \sim T'_2\end{array}}{cx \vdash T_1 \to T_2 \sim T'_1 \to T'_2}\quad(\textsc{TeArr})$$

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \forall i.\ \ cx \vdash T_i \sim T'_i\end{array}}{cx \vdash \prod_i f_i\ T_i \sim \prod_i f_i\ T'_i}\quad(\textsc{TeProd})$$

$$\frac{\forall i.\ \ cx \vdash T_i \sim T'_i}{cx \vdash \sum_i c_i\ T_i \sim \sum_i f_i\ T'_i} \quad (\textsc{TeSum})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \textsc{type} \\ cx \vdash T \sim T' \end{array}}{cx \vdash T|r \sim T'|r} \quad (\textsc{TeSub})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ cx \vdash T \sim T' \end{array}}{cx \vdash T/q \sim T'/q} \quad (\textsc{TeQuot})$$

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

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

Explanation:

- A type definition introduces type equivalences, one for each instance of the defining equation.
- Type equivalence is reflexive, symmetric, and transitive, i.e. it is indeed an equivalence relation.
- Type equivalence is a congruence with respect to syntactic (meta-)operations to construct types in the Metaslang type system, namely type instantiations, arrow types, product types, sum types, subtypes, and quotient types.
- Equivalence of subtypes and quotient types as defined above is stronger than necessary: weaker rules would make e.g. $T|r$ and $T'|r'$ equivalent if T and T' are equivalent and r and r' are provably equal (in the Metaslang logic). The definition above requires the subtype and quotient type predicates to be syntactically the same for two subtypes or quotient types to be equivalent.
- The last two rules say that the order of the factors of a product type and the order of the summands of a sum type is unimportant, because any permutation of the factors or summands yields equivalent types. The permutation is captured by a bijective function P on the product or sum indices $\{1,\ldots,n\}$ (the rules explicitly say that P is injective, but since domain and codomain are finite and equal, it follows that P is also surjective, hence bijective).

3.4 Well-typed expressions

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

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

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

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

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

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

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

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

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

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

$$\frac{cx \vdash T|r : \textsc{type}}{cx \vdash \mathsf{rel}_r : T|r \rightarrow T} \quad (\textsc{exRelax})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \textsc{type} \\ cx \vdash e : T \\ cx \vdash r\ e \end{array}}{cx \vdash \mathsf{res}_r\ e : T|r} \quad (\textsc{exRestr})$$

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

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

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

$$\frac{\begin{array}{c} cx \vdash \exists!\, v\!:\!T.\ v \equiv e \\ cx, \mathsf{var}\ v\!:\!T \vdash e' : T' \end{array}}{cx \vdash \mathsf{letr}\ v\!:\!T \!\leftarrow\! e\ \mathsf{in}\ e' : T'} \quad (\textsc{exLetRec})$$

$$\frac{\begin{array}{c} cx \vdash e : T \\ cx \vdash T \sim T' \end{array}}{cx \vdash e : T'} \quad (\textsc{exTE})$$

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 $\overline{\alpha}$. The result is a well-formed expression whose type is obtained by substituting $\overline{\alpha}$ 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 restriction to be well-typed, the argument expression must satisfy the predicate that defines the the associated subtype.

- 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'.

- 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, tuples, projections, embedders, relaxators, and quotienters are straightforward.

- The last rule, exTE, links the notion of well-typed expressions with the notion of type equivalence: if an expression e has a type T, it also has any type T' equivalent to T.

3.5 Well-typed patterns

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

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

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

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

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

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

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 tuple patterns are straightforward.
- The last rule, paTE, links the notion of well-typed patterns with the notion of type equivalence: if a pattern p has a type T, it also has any type T' equivalent to T.

3.6 Theorems

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

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

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

$$\frac{\begin{array}{c}cx \vdash e : \mathsf{Bool} \to \mathsf{Bool} \\ v \in \mathcal{N} - \mathcal{V}(cx)\end{array}}{cx \vdash e\ \mathsf{true} \wedge e\ \mathsf{false} \Leftrightarrow (\forall v{:}\mathsf{Bool}.\ e\ v)} \quad (\textsc{thBool})$$

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

$$\frac{\begin{array}{c}cx \vdash e_1 : T \to T' \\ cx \vdash e_2 : T \to T' \\ v \in \mathcal{N} - \mathcal{V}(cx)\end{array}}{cx \vdash e_1 \equiv e_2 \Leftrightarrow (\forall v{:}T.\ e_1\ v \equiv e_2\ v)} \quad (\textsc{thExt})$$

$$\frac{cx \vdash (\lambda v{:}T.e)\ e' : T'}{cx \vdash (\lambda v{:}T.e)\ e' \equiv e[e'/v]} \quad (\textsc{thAbs})$$

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

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

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \text{TYPE} \\ v, \overline{v} \in \mathcal{N}^{(*)} \\ v, \overline{v} \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}\prod_i f_i\ T_i.\ (\exists \overline{v}{:}\overline{T}.\ v \equiv \{f_i \leftarrow v_i\}_i)} \quad (\textsc{thTuple})$$

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

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \text{TYPE} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}\sum_i c_i\ T_i.\ \bigvee_i \exists v'{:}T_i.\ v \equiv \mathsf{emb}\ c_i\ v'} \quad (\textsc{thEmbSurj})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \text{TYPE} \\ j \neq k \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}T_j, v'{:}T_k.\ \mathsf{emb}\ c_j\ v \not\equiv \mathsf{emb}\ c_k\ v'} \quad (\textsc{thEmbDist})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \text{TYPE} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}T_j, v'{:}T_j.\ v \not\equiv v' \Rightarrow \mathsf{emb}\ c_j\ v \not\equiv \mathsf{emb}\ c_j\ v'} \quad (\textsc{thEmbInj})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \text{TYPE} \\ v \in \mathcal{N} - \mathcal{V}(cx) \end{array}}{cx \vdash \forall v{:}T|r.\ r\ (\mathsf{rel}_r\ v)} \quad (\textsc{thRlxPred})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \text{TYPE} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}T|r, v'{:}T|r.\ v \not\equiv v' \Rightarrow \mathsf{rel}_r\ v \not\equiv \mathsf{rel}_r\ v'} \quad (\textsc{thRlxInj})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \text{TYPE} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v{:}T.\ r\ v \Rightarrow (\exists v'{:}T|r.\ v \equiv \mathsf{rel}_r\ v')} \quad (\textsc{thRlxSurj})$$

$$\frac{\begin{array}{c} cx \vdash T|r : \textsc{type} \\ v \in \mathcal{N} - \mathcal{V}(cx) \end{array}}{cx \vdash \forall v\!:\!T|r.\ \mathsf{res}_r\ (\mathsf{rel}_r\ v) \equiv v} \quad (\textsc{thRestr})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v\!:\!T/q.\ (\exists v'\!:\!T.\ \mathsf{quo}_q\ v' \equiv v)} \quad (\textsc{thQuotSurj})$$

$$\frac{\begin{array}{c} cx \vdash T/q : \textsc{type} \\ v, v' \in \mathcal{N}^{(*)} \\ v, v' \cap \mathcal{V}(cx) = \emptyset \end{array}}{cx \vdash \forall v\!:\!T, v'\!:\!T.\ q\ \langle v, v' \rangle \Leftrightarrow \mathsf{quo}_q\ v \equiv \mathsf{quo}_q\ v'} \quad (\textsc{thQuotEqCls})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{ch}_q\ e : T/q \to T' \\ v \in \mathcal{N} - \mathcal{V}(cx) \end{array}}{cx \vdash \forall v\!:\!T.\ (\mathsf{ch}_q\ e)\ (\mathsf{quo}_q\ v) \equiv e\ v} \quad (\textsc{thChoose})$$

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

$$\frac{\begin{array}{c} cx \vdash \mathsf{letr}\ v\!:\!T \!\leftarrow\! e\ \mathsf{in}\ e' : T' \\ cx, \mathsf{var}\ v\!:\!T, \mathsf{ax}\ v \equiv e \vdash e' \equiv e'' \\ v \notin \mathcal{FV}(e'') \end{array}}{cx \vdash \mathsf{letr}\ v\!:\!T \!\leftarrow\! e\ \mathsf{in}\ e' \equiv e''} \quad (\textsc{thLetRec})$$

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 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 be already declared in the context.

- 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 domains, 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 thRepl 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.
- 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 thTuple and thProj characterize product types. The first rule says that every value of a product type is a tuple; 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 tuple construction, viewed as a function, to be injective, because if tuples with different arguments were mapped to the same value of the product type, the theorem asserted by the rule would be violated.
- 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 thRlxPred, thRlxInj, and thRlxSurj characterize subtypes. They say that rel_r is a bijective function from the subtype to the subset of the supertype values that satisfy the subtype predicate r.
- Rule thRestr defines the semantics of restrictions as left inverses of the associated relaxators.
- 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 THLETREC defines the semantics of recursive let's. The context is extended with the equality derived by the binding and if the body e' is provably equal to an expression e'' where the let-bound variable v does not occur free, the whole recursive let is provably equal to e''. The requirement that v does not occur free in e'' ensures that e'' is well-typed in the unextended context in which the recursive let is well-typed.

3.7 Proofs

The previous subsections have defined assertions of the forms

$$\begin{array}{l} \vdash cx : \text{CONTEXT} \\ cx \vdash T : \text{TYPE} \\ cx \vdash T_1 \sim T_2 \\ cx \vdash e : T \\ cx \vdash p : T \\ cx \vdash e \end{array}$$

by means of a set of inductive rules.

A proof of an assertion is a finite sequence of assertions that ends with the proved assertion and where each assertion in the sequence is derived from preceding assertions 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 assertion $cx \vdash e : T$ can be derived then also $cx \vdash T : \text{TYPE}$ 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. The properties also concern free and bound variables, e.g. that certain substitutions do not cause variable capture (if that turns out to not the case, suitable conditions will be added to the rules in §3.

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 $T_1 \to T_2$ 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.