The Logic of Metaslang

Alessandro Coglio

May 6, 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 is possible to define the Metaslang logic in axiomatic set theory or any other sufficiently expressive formal language.

The (meta-)logical notations $=$, $\forall$, $\wedge$, 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. $\{0, 1, 2, \ldots\}$.

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, $A \xrightarrow{\mathrm{p}} B$ 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. A sequence $x_1, \ldots, x_n$ is often written $\overline{x}$, leaving n implicit. 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

$$\mathcal{N}$$

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 must match its arity). We may write $\tau[\epsilon]$ as just τ.
- An arrow type $T_1 \to T_2$ consists of a domain T_1 and a range 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 may have no fields (denoted $\prod \epsilon$), while sum types always have constructors. 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 $\prod \epsilon$ where a constructor has no type, and add the empty record as argument to c_i in expressions and patterns where needed.
- Restriction types $T|r$ and quotient types T/q are obtained by combining types T with expressions r and q (meant to be suitable predicates). Restriction and quotient types create the dependency of types on expressions. Restriction types as defined above also capture comprehension types as defined in [1]: as mentioned in [1], a comprehension type can be turned into a restriction type by re-combining the pattern and expression into a lambda expression.

We introduce the abbreviation

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

where each π_i is a fixed but unspecified name in $\mathcal{N}$ such that $\pi_i \neq \pi_j$ if $i \neq j$. Thus, record types as defined above also capture product types as defined in [1]; the names π_i capture the natural literal fields used in [1].

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

2.3 Expressions

We inductively define the set of expressions as

$$\begin{aligned}
\mathit{Exp} = {} & \{v \mid v \in \mathcal{N}\} \\
& + \{o[\overline{T}] \mid o \in \mathcal{N} \ \wedge \ \overline{T} \in \mathit{Type}^*\} \\
& + \{e_1\ e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\lambda v\!:\!T.\ e \mid v \in \mathcal{N} \ \wedge \ T \in \mathit{Type} \ \wedge \ e \in \mathit{Exp}\} \\
& + \{e_1 \equiv e_2 \mid e_1, e_2 \in \mathit{Exp}\} \\
& + \{\mathsf{if}\ e_0\ e_1\ e_2 \mid e_0, e_1, e_2 \in \mathit{Exp}\} \\
& + \{\iota_T \mid T \in \mathit{Type}\} \\
& + \{\mathsf{proj}_{\prod_i f_i\ T_i}\ f_j \mid \textstyle\prod_i f_i\ T_i \in \mathit{Type}\} \\
& + \{\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j \mid \textstyle\sum_i c_i\ T_i \in \mathit{Type}\} \\
& + \{\mathsf{quo}_{T/q} \mid T/q \in \mathit{Type}\}
\end{aligned}$$

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 $\overline{T}$ that instantiate the (generally, polymorphic) type of the declared op. We may write $o[\epsilon]$ as just o.
- An application $e_1\ e_2$ consists of a function e_1 juxtaposed to an argument e_2.
- A (lambda) abstraction $\lambda v\!:\!T.\ e$ consists of an argument v with an explicit type T and a body e. Even though lambda expressions as defined in [1] may have branches with patterns, that does not increase expressivity: one can imagine to use a fresh variable as argument of the abstraction and a case expression (introduced later) on the fresh variable with the branches as the body.
- An equality $e_1 \equiv e_2$ consists of a left-hand side e_1 and a right-hand side e_2.
- A conditional $\mathsf{if}\ e_0\ e_1\ e_2$ consists of a condition e_0, a "then" branch e_1, and an "else" branch e_2.
- The description operator ι_T, currently absent from [1], is tagged by a type. It operates on predicates over T (i.e. over values of type $T \rightarrow \mathsf{Bool}$) that are satisfied by a unique value of T, and its result is *the* value that satisfies the predicate.
- A projector $\mathsf{proj}_{\prod_i f_i\ T_i}\ f_j$ is tagged by a record type and by a field of that record type. We may write $\mathsf{proj}_{\prod_i f_i\ T_i}\ f_j$ as just $\mathsf{proj}\ f_j$ when the record type is inferrable or irrelevant.
- An embedder $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j$ is tagged by a sum type and by a constructor of that sum type. We may write $\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j$ as just $\mathsf{emb}\ c_j$ when the sum type is inferrable or irrelevant.
- A quotienter $\mathsf{quo}_{T/q}$ is tagged by a quotient type. We may write $\mathsf{quo}_{T/q}$ as just quo when the quotient type is inferrable or irrelevant.

We introduce the abbreviations (many of which are expressions defined in [1])

$$
\begin{array}{rcl}
\mathsf{true} & \longrightarrow & \lambda\gamma:\mathsf{Bool}.\ \gamma \equiv \lambda\gamma:\mathsf{Bool}.\ \gamma \\
\mathsf{false} & \longrightarrow & \lambda\gamma:\mathsf{Bool}.\ \gamma \equiv \lambda\gamma:\mathsf{Bool}.\ \mathsf{true} \\
\neg & \longrightarrow & \lambda\gamma:\mathsf{Bool}.\ (\mathsf{if}\ \gamma\ \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} \\
\Leftrightarrow & \longrightarrow & \lambda\gamma:\mathsf{Bool}.\ \lambda\gamma':\mathsf{Bool}.\ (\gamma \equiv \gamma') \\
e_1 \Leftrightarrow e_2 & \longrightarrow & \Leftrightarrow\ e_1\ e_2 \\
e_1 \not\equiv e_2 & \longrightarrow & \neg\ (e_1 \equiv e_2) \\
\iota v:T.e & \longrightarrow & \iota_T\ (\lambda v:T.\ e) \\
\forall_T & \longrightarrow & \lambda\psi:T \to \mathsf{Bool}.\ (\psi \equiv \lambda\gamma:T.\ \mathsf{true}) \\
\forall v:T.\ e & \longrightarrow & \forall_T\ (\lambda v:T.\ e) \\
\forall v_1:T_1,\ldots,v_n:T_n.\ e & \longrightarrow & \forall v_1:T_1.\ \ldots \forall v_n:T_n.\ e \\
\forall \overline{v}:\overline{T}.\ e & \longrightarrow & \forall v_1:T_1,\ldots,v_n:T_n.\ e \\
\exists_T & \longrightarrow & \lambda\psi:T \to \mathsf{Bool}.\ \neg\ (\forall\gamma:T.\ \neg\ (\psi\ \gamma)) \\
\exists v:T.\ e & \longrightarrow & \exists_T\ (\lambda v:T.\ 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 \\
\exists \overline{v}:\overline{T}.\ e & \longrightarrow & \exists v_1:T_1,\ldots,v_n:T_n.\ e \\
\exists!_T & \longrightarrow & \lambda\psi:T \to \mathsf{Bool}.\ (\exists\gamma:T.\ (\psi\ \gamma \wedge \forall\gamma':T.\ (\psi\ \gamma' \Rightarrow \gamma' \equiv \gamma))) \\
\exists!\, v:T.\ e & \longrightarrow & \exists!_T\ (\lambda v:T.\ e) \\
e.f & \longrightarrow & \mathsf{proj}\ f\ e \\
\mathsf{rec}_{\prod_i f_i\ T_i} & \longrightarrow & \lambda\gamma_1:T_1.\ \ldots \lambda\gamma_n:T_n.\ \iota\gamma:\prod_i f_i\ T_i.\ \bigwedge_i(\gamma.f_i \equiv \gamma i) \\
\langle f_1 \overset{T_1}{\leftarrow} e_1\ \ldots\ f_n \overset{T_n}{\leftarrow} e_n\rangle & \longrightarrow & \mathsf{rec}_{\prod_i f_i\ T_i}\ e_1\ \ldots\ e_n \\
\langle e_1,\ldots,e_n\rangle & \longrightarrow & \langle \pi_1 \leftarrow e_1\ \ldots\ \pi_n \leftarrow e_n\rangle
\end{array}
$$

where γ, γ', ψ, and each γ_i are fixed but unspecified names in $\mathcal{N}$ such that they are all distinct.

Explanation:

- The abbreviations true and false stand for logical truth and falsehood, respectively.
- The logical connectives for negation, conjunction, disjunction, and implication are defined in terms of conditionals.
- Coimplication is a synonym for equality, but only for booleans.
- Inequality is negation of equality.
- Stand-alone quantifiers are higher-order functions, but they can be written in binder form. Also the description operator can be written in binder form. Note that both $\forall\epsilon.\ e$ and $\exists\epsilon.\ e$ abbreviate e.
- A dotted projection $e.f$ abbreviates an applied projection whose record type is implicit; this is why no record type appears in $e.f$.
- A record constructor $\mathsf{rec}_{\prod_i f_i\ T_i}$ is tagged by a record type and is defined by means of a description; records are characterized by their components. The notation $\langle f_i \overset{T_i}{\leftarrow} e_i\rangle_i$ is a more readable version of an applied record constructor. We may write $\langle f_i \overset{T_i}{\leftarrow} e_i\rangle_i$ as just $\langle f_i \leftarrow e_i\rangle_i$ when the component types are inferrable or irrelevant.
- A tuple $\langle\overline{e}\rangle$ captures tuple displays as defined in [1]. It abbreviates a record construction of a product type whose component types are implicit; this is why no component types appear in $\langle\overline{e}\rangle$.

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

$$\begin{aligned}
\mathcal{FV}(v) &= \{v\} \\
\mathcal{FV}(o[\overline{T}]) &= \emptyset \\
\mathcal{FV}(e_1\ e_2) &= \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\lambda v\!:\!T.\ e) &= \mathcal{FV}(e) - \{v\} \\
\mathcal{FV}(e_1 \equiv e_2) &= \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\mathsf{if}\ e_0\ e_1\ e_2) &= \mathcal{FV}(e_0) \cup \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2) \\
\mathcal{FV}(\iota_T) &= \emptyset \\
\mathcal{FV}(\mathsf{proj}\ f) &= \emptyset \\
\mathcal{FV}(\mathsf{emb}\ c) &= \emptyset \\
\mathcal{FV}(\mathsf{quo}) &= \emptyset
\end{aligned}$$

Note that we do not consider the free variables in expressions contained in types contained in expressions (e.g. we do not consider the free variables in q as part of the free variables of $\mathsf{quo}_{T/q}$). The reason is that, as defined later, all the expressions contained in well-formed types have no free variables.

2.4 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 write $\mathsf{ax}\ \{\epsilon\}\ e$ as just $\mathsf{ax}\ e$.
- A type variable declaration $\mathsf{tvar}\ \beta$ introduces a type variable. We may write $\mathsf{tvar}\ \beta_1, \ldots, \mathsf{tvar}\ \beta_n$ as just $\mathsf{tvar}\ \beta_1, \ldots, \beta_n$.
- A variable declaration $\mathsf{var}\ v\!:\!T$ introduces a variable with a type.

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

$$\begin{array}{ll} \mathcal{TN}(\epsilon) & = \emptyset \\ \mathcal{TN}(cxel, cx) & = \begin{cases} \mathcal{TN}(cx) \cup \{\tau\} & \text{if} \quad cxel = \mathsf{ty}\ \tau : n \\ \mathcal{TN}(cx) & \text{otherwise} \end{cases} \end{array}$$

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

$$\begin{array}{ll} \mathcal{ON}(\epsilon) & = \emptyset \\ \mathcal{ON}(cxel, cx) & = \begin{cases} \mathcal{ON}(cx) \cup \{o\} & \text{if} \quad cxel = \mathsf{op}\ o : \{\overline{\beta}\}\ T \\ \mathcal{ON}(cx) & \text{otherwise} \end{cases} \end{array}$$

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

$$\begin{array}{ll} \mathcal{TV}(\epsilon) & = \emptyset \\ \mathcal{TV}(cxel, cx) & = \begin{cases} \mathcal{TV}(cx) \cup \{\beta\} & \text{if} \quad cxel = \mathsf{tvar}\ \beta \\ \mathcal{TV}(cx) & \text{otherwise} \end{cases} \end{array}$$

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

$$\begin{array}{ll} \mathcal{V}(\epsilon) & = \emptyset \\ \mathcal{V}(cxel, cx) & = \begin{cases} \mathcal{V}(cx) \cup \{v\} & \text{if} \quad cxel = \mathsf{var}\ v : T \\ \mathcal{V}(cx) & \text{otherwise} \end{cases} \end{array}$$

2.5 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.

2.6 Occurring ops

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

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

$$\begin{array}{ll} \mathcal{ON}(v) & = \emptyset \\ \mathcal{ON}(o[\overline{T}]) & = \{o\} \cup \bigcup_i \mathcal{ON}(T_i) \\ \mathcal{ON}(e_1\ e_2) & = \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\ \mathcal{ON}(\lambda v : T.\ e) & = \mathcal{ON}(T) \cup \mathcal{ON}(e) \\ \mathcal{ON}(e_1 \equiv e_2) & = \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\ \mathcal{ON}(\mathsf{if}\ e_0\ e_1\ e_2) & = \mathcal{ON}(e_0) \cup \mathcal{ON}(e_1) \cup \mathcal{ON}(e_2) \\ \mathcal{ON}(\iota_T) & = \mathcal{ON}(T) \\ \mathcal{ON}(\mathsf{proj}_{\prod_i f_i\ T_i}\ f_j) & = \mathcal{ON}(\prod_i f_i\ T_i) \\ \mathcal{ON}(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j) & = \mathcal{ON}(\sum_i c_i\ T_i) \\ \mathcal{ON}(\mathsf{quo}_{T/q}) & = \mathcal{ON}(T/q) \end{array}$$

2.7 Substitutions

2.7.1 Type substitutions

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

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

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

$$\begin{aligned}
v[\overline{\beta}/\overline{S}] &= v \\
o[\overline{T}][\overline{\beta}/\overline{S}] &= o[\overline{T}[\overline{\beta}/\overline{S}]] \\
(e_1\ e_2)[\overline{\beta}/\overline{S}] &= e_1[\overline{\beta}/\overline{S}]\ e_2[\overline{\beta}/\overline{S}] \\
(\lambda v{:}T.\ e)[\overline{\beta}/\overline{S}] &= \lambda v{:}T[\overline{\beta}/\overline{S}].\ e[\overline{\beta}/\overline{S}] \\
(e_1 \equiv e_2)[\overline{\beta}/\overline{S}] &= e_1[\overline{\beta}/\overline{S}] \equiv e_2[\overline{\beta}/\overline{S}] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[\overline{\beta}/\overline{S}] &= \mathsf{if}\ e_0[\overline{\beta}/\overline{S}]\ e_1[\overline{\beta}/\overline{S}]\ e_2[\overline{\beta}/\overline{S}] \\
(\iota_T)[\overline{\beta}/\overline{S}] &= \iota_{T[\overline{\beta}/\overline{S}]} \\
\left(\mathsf{proj}_{\prod_i f_i\ T_i}\ f_j\right)[\overline{\beta}/\overline{S}] &= \mathsf{proj}_{(\prod_i f_i\ T_i)[\overline{\beta}/\overline{S}]}\ f_j \\
\left(\mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\right)[\overline{\beta}/\overline{S}] &= \mathsf{emb}_{(\sum_i c_i\ T_i)[\overline{\beta}/\overline{S}]}\ c_j \\
\left(\mathsf{quo}_{T/q}\right)[\overline{\beta}/\overline{S}] &= \mathsf{quo}_{(T/q)[\overline{\beta}/\overline{S}]}
\end{aligned}$$

2.7.2 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 $e[u/d]$)

$$\begin{aligned}
v[u/d] &= \begin{cases} d & \text{if} \quad u = v \\ v & \text{otherwise} \end{cases} \\
o[\overline{T}][u/d] &= o[\overline{T}] \\
(e_1\ e_2)[u/d] &= e_1[u/d]\ e_2[u/d] \\
(\lambda v{:}T.\ e)[u/d] &= \begin{cases} \lambda v{:}T.\ e & \text{if} \quad u = v \\ \lambda v{:}T.\ e[u/d] & \text{otherwise} \end{cases} \\
(e_1 \equiv e_2)[u/d] &= e_1[u/d] \equiv e_2[u/d] \\
(\mathsf{if}\ e_0\ e_1\ e_2)[u/d] &= \mathsf{if}\ e_0[u/d]\ e_1[u/d]\ e_2[u/d] \\
(\iota_T)[u/d] &= \iota_T \\
(\mathsf{proj}\ f)[u/d] &= \mathsf{proj}\ f \\
(\mathsf{emb}\ c)[u/d] &= \mathsf{emb}\ c \\
\mathsf{quo}[u/d] &= \mathsf{quo}
\end{aligned}$$

No substitution is performed in the expressions contained in types contained in expressions because, as already mentioned, such inner expressions have no free variables in well-formed types.

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)

$$
\begin{array}{ll}
\mathcal{CV}(v,u) & = \emptyset \\
\mathcal{CV}(o[\overline{T}],u) & = \emptyset \\
\mathcal{CV}(e_1\ e_2,u) & = \mathcal{CV}(e_1,u) \cup \mathcal{CV}(e_2,u) \\
\mathcal{CV}(\lambda v\!:\!T.\ e,u) & = \begin{cases} \{v\} \cup \mathcal{CV}(e,u) & \text{if} \quad u \in \mathcal{FV}(e) \ \wedge \ u \neq v \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{CV}(e_1 \equiv e_2,u) & = \mathcal{CV}(e_1,u) \cup \mathcal{CV}(e_2,u) \\
\mathcal{CV}(\mathsf{if}\ e_0\ e_1\ e_2,u) & = \mathcal{CV}(e_0,u) \cup \mathcal{CV}(e_1,u) \cup \mathcal{CV}(e_2,u) \\
\mathcal{CV}(\iota_T,u) & = \emptyset \\
\mathcal{CV}(\mathsf{proj}\ f,u) & = \emptyset \\
\mathcal{CV}(\mathsf{emb}\ c,u) & = \emptyset \\
\mathcal{CV}(\mathsf{quo},u) & = \emptyset
\end{array}
$$

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

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

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 $\vdash \_ : \textsc{context} \subseteq \mathit{Cx}$ to capture well-formed contexts as

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

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

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

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

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

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

$$\frac{\begin{array}{c}\vdash cx : \textsc{context} \\ \beta \notin \mathcal{TV}(cx)\end{array}}{\vdash cx, \mathsf{tvar}\ \beta : \textsc{context}} \quad (\textsc{cxTVdec})$$

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

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 $\mathsf{op}\ o{:}\{\overline{\beta}\}\ T$ 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 $\overline{\beta}'$.
- A 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

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

Note that the unary relation $\vdash \_ : \textsc{spec}$ is defined on set Sp, which, as previously defined, consists of all the contexts without type variable and variable declarations. Thus, there is no need to include, as part of this rule, the condition that sp does not contain any type variable and variable declaration. The rule just says that a spec (which has no type variable or variable declarations by definition) is well-formed as a spec if it is well-formed as a context.

3.3 Well-formed types

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

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

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

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

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

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

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

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

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

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 restriction types, 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.
- 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). 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. We require that q has no free variables.

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

$$\frac{\begin{array}{c}\vdash cx : \text{CONTEXT} \\ \mathsf{def}\ \tau[\overline{\beta}] = T \in cx \\ \forall i.\ \ cx \vdash T_i : \text{TYPE}\end{array}}{cx \vdash \tau[\overline{T}] \approx T[\overline{\beta}/\overline{T}]} \quad (\text{TEDEF})$$

$$\frac{cx \vdash T : \text{TYPE}}{cx \vdash T \approx T} \quad (\text{TEREFL})$$

$$\frac{cx \vdash T_1 \approx T_2}{cx \vdash T_2 \approx T_1} \quad (\text{TESYMM})$$

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

$$\frac{\begin{array}{c}cx \vdash \tau[\overline{T}] : \text{TYPE} \\ \forall i.\ \ cx \vdash T_i \approx T'_i\end{array}}{cx \vdash \tau[\overline{T}] \approx \tau[\overline{T}']} \quad (\text{TEINST})$$

$$\frac{\begin{array}{c}cx \vdash T_1 \approx T'_1 \\ cx \vdash T_2 \approx T'_2\end{array}}{cx \vdash T_1 \to T_2 \approx T'_1 \to T'_2} \quad (\text{TEARR})$$

$$\frac{\begin{array}{c}\vdash cx : \text{CONTEXT} \\ \forall i.\ \ cx \vdash T_i \approx T'_i\end{array}}{cx \vdash \prod_i f_i\ T_i \approx \prod_i f_i\ T'_i} \quad (\text{TEREC})$$

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

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

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

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

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

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, restriction types, and quotient types. In addition, equal restriction or quotient predicates give rise to equivalent restriction or quotient types.
- The order of the components of a record or sum type is unimportant: 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

$$\frac{cx \vdash T|r : \textsc{type}}{cx \vdash T|r \prec_r T} \quad (\textsc{stRestr})$$

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

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

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \text{TYPE} \\ \forall i.\ \ cx \vdash T_i \prec_{r_i} T'_i \\ r = \lambda v{:}\prod_i f_i\ T'_i.\ \bigwedge_i r_i\ v.f_i \end{array}}{cx \vdash \prod_i f_i\ T_i \prec_r \prod_i f_i\ T'_i} \quad (\textsc{stRec})$$

$$\frac{\begin{array}{c} cx \vdash \sum_i c_i\ T_i : \text{TYPE} \\ \forall i.\ \ cx \vdash T_i \prec_{r_i} T'_i \\ r = \lambda v{:}\sum_i c_i\ T'_i.\ \bigvee_i \exists v'{:}T'_i.\ (v \equiv \mathsf{emb}\ c_i\ v' \wedge r_i\ v') \\ v \neq v' \end{array}}{cx \vdash \sum_i c_i\ T_i \prec_r \sum_i c_i\ T'_i} \quad (\textsc{stSum})$$

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

Explanation:

- Unsurprisingly, a restriction type $T|r$ 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.
- Rule stTE states the substitutivity of equivalent types in subtype 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

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

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

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

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

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

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

$$\frac{cx \vdash T : \textsc{type}}{cx \vdash \iota_T : (T \to \mathsf{Bool})|\exists!_T \to T} \quad (\textsc{exThe})$$

$$\frac{cx \vdash \prod_i f_i\ T_i : \textsc{type}}{cx \vdash \mathsf{proj}\ f_j : \prod_i f_i\ T_i \to T_j} \quad (\textsc{exProj})$$

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

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

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

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

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

Explanation:

- A variable v declared in a well-formed context is a well-typed expression with the type T given in its declaration.
- 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.
- An application is well-typed if the function has an arrow type and the argument has the domain type of the arrow type. The type of the application is the range type of the arrow type.
- An abstraction is well-typed if the body is well-typed in the context extended with the declaration of the variable bound by the abstraction. The type of the abstraction is the arrow type that has the type of the variable as domain and the type of the body as range.
- An equality is well-typed with type Bool if the left- and right-hand sides are both well-typed with a common type T.
- In the rule exIf for conditionals, the two branches must be well-typed, with a common type, 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. This rule makes conditionals non-strict.
- The description operator for a well-formed type is well-typed and denotes a function from the predicates over the type that are satisfied by a unique value to the type itself.
- A projector for a well-formed record type is well-typed and denotes a function from the record type to the corresponding component type.
- An embedded for a well-formed sum type is well-typed and denotes a function from the type corresponding to the constructor to the sum type.
- A quotienter for a well-formed quotient type is well-typed and denotes a function from the quotiented type to the quotient type.
- 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 rule amounts to treating expressions up to alpha equivalence, allowing to rename bound variables maintaining well-typedness. Without this rule, 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). This rule also allows variable hiding as described in [1].

3.7 Theorems

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

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

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

$$\frac{cx \vdash e : \ldots}{cx \vdash e \equiv e} \quad (\textsc{thRefl})$$

$$\frac{cx \vdash e_1 \equiv e_2}{cx \vdash e_2 \equiv e_1} \quad (\textsc{thSymm})$$

$$\frac{\begin{array}{c}cx \vdash e_1 \equiv e_2 \\ cx \vdash e_2 \equiv e_3\end{array}}{cx \vdash e_1 \equiv e_3} \quad (\textsc{thTrans})$$

$$\frac{\begin{array}{c}cx \vdash o[\overline{T}] : \ldots \\ \forall i.\ \ cx \vdash T_i \approx T_i'\end{array}}{cx \vdash o[\overline{T}] \equiv o[\overline{T}']} \quad (\textsc{thOpSubst})$$

$$\frac{\begin{array}{c}cx \vdash e_1\ e_2 : \ldots \\ cx \vdash e_1 \equiv e_1' \\ cx \vdash e_2 \equiv e_2'\end{array}}{cx \vdash e_1\ e_2 \equiv e_1'\ e_2'} \quad (\textsc{thAppSubst})$$

$$\frac{\begin{array}{c}cx \vdash \lambda v\!:\!T.\ e : \ldots \\ cx \vdash T \approx T' \\ cx, \mathsf{var}\ v\!:\!T \vdash e \equiv e'\end{array}}{cx \vdash \lambda v\!:\!T.\ e \equiv \lambda v\!:\!T'.\ e'} \quad (\textsc{thAbsSubst})$$

$$\frac{\begin{array}{c}cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 : \ldots \\ cx, \mathsf{ax}\ e_0 \vdash e_1 \equiv e_1' \\ cx, \mathsf{ax}\ \neg\ e_0 \vdash e_2 \equiv e_2'\end{array}}{cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 \equiv \mathsf{if}\ e_0\ e_1'\ e_2'} \quad (\textsc{thIfSubst})$$

$$\frac{\begin{array}{c}cx \vdash \iota_T : \ldots \\ cx \vdash T \approx T'\end{array}}{cx \vdash \iota_T \equiv \iota_{T'}} \quad (\textsc{thTheSubst})$$

$$\frac{\begin{array}{c}cx \vdash \mathsf{proj}_{\prod_i f_i\ T_i}\ f : \ldots \\ cx \vdash \prod_i f_i\ T_i \approx \prod_i f_i'\ T_i'\end{array}}{cx \vdash \mathsf{proj}_{\prod_i f_i\ T_i}\ f \equiv \mathsf{proj}_{\prod_i f_i'\ T_i'}\ f} \quad (\textsc{thProjSubst})$$

$$\frac{\begin{array}{c}cx \vdash \mathsf{emb}_{\sum_i c_i\ T_i}\ c : \ldots \\ cx \vdash \sum_i c_i\ T_i \approx \sum_i c_i'\ T_i'\end{array}}{cx \vdash \mathsf{emb}_{\sum_i c_i\ T_i}\ c \equiv \mathsf{emb}_{\sum_i c_i'\ T_i'}\ c} \quad (\textsc{thEmbedSubst})$$

$$\frac{\begin{array}{c} cx \vdash \mathsf{quo}_{T/q} : \ldots \\ cx \vdash T/q \approx T'/q' \end{array}}{cx \vdash \mathsf{quo}_{T/q} \equiv \mathsf{quo}_{T'/q'}} \quad (\textsc{thQuotSubst})$$

$$\frac{\begin{array}{c} cx \vdash e \\ cx \vdash e \equiv e' \end{array}}{cx \vdash e'} \quad (\textsc{thSubst})$$

$$\frac{\begin{array}{c} \vdash cx : \textsc{context} \\ v \neq v' \end{array}}{cx \vdash \forall v\!:\!\mathsf{Bool} \to \mathsf{Bool}.\ (v\ \mathsf{true} \wedge v\ \mathsf{false} \Leftrightarrow (\forall v'\!:\!\mathsf{Bool}.\ v\ v'))} \quad (\textsc{thBool})$$

$$\frac{\begin{array}{c} cx \vdash T \to T' : \textsc{type} \\ v \neq v' \ \wedge\ v' \neq v'' \ \wedge\ v'' \neq v \end{array}}{cx \vdash \forall v\!:\!T \to T', v'\!:\!T \to T'.\ (v \equiv v' \Leftrightarrow (\forall v''\!:\!T.\ v\ v'' \equiv v'\ v''))} \quad (\textsc{thExt})$$

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

$$\frac{\begin{array}{c} cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 : \ldots \\ cx, \mathsf{ax}\ e_0 \vdash e_1 \equiv e' \\ cx, \mathsf{ax}\ \neg\ e_0 \vdash e_2 \equiv e' \end{array}}{cx \vdash \mathsf{if}\ e_0\ e_1\ e_2 \equiv e'} \quad (\textsc{thIf})$$

$$\frac{cx \vdash \iota_T\ e : T}{cx \vdash e\ (\iota_T\ e)} \quad (\textsc{thThe})$$

$$\frac{\begin{array}{c} cx \vdash \prod_i f_i\ T_i : \textsc{type} \\ v \neq v' \end{array}}{\forall v\!:\!\prod_i f_i\ T_i, v'\!:\!\prod_i f_i\ T_i.\ ((\bigwedge_i v.f_i \equiv v'.f_i) \Rightarrow v \equiv v')} \quad (\textsc{thRec})$$

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

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

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

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

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

$$\frac{cx \vdash \prod_i f_i\ T_i \prec_r \prod_i f_i\ T'_i}{cx \vdash \forall v\!:\!\prod_i f_i\ T_i.\ \mathsf{proj}_{\prod_i f_i\ T_i}\ f_j\ v \equiv \mathsf{proj}_{\prod_i f_i\ T'_i}\ f_i\ v} \quad (\textsc{thProjSub})$$

$$\frac{cx \vdash \sum_i c_i\ T_i \prec_r \sum_i c_i\ T'_i}{cx \vdash \forall v\!:\!T_j.\ \mathsf{emb}_{\sum_i c_i\ T_i}\ c_j\ v \equiv \mathsf{emb}_{\sum_i c_i\ T'_i}\ c_j\ v} \quad (\textsc{thEmbSub})$$

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

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.
- Rules thRefl, thSymm, and thTrans say that equality is an equivalence, i.e. reflexive, symmetric, and transitive.
- Rules thOpSubst to thQuotSubst state the substitutivity of equalities and type equivalences in expressions. Note that in rule thAbsSubst the context is extended with the variable bound by the abstraction; in rule thIfSubst the context is extended with an axiom saying that the condition is true or false.
- Rule thSubst says that anything equal to a theorem is itself a theorem.
- 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.
- 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.
- Rule thThe says that a description satisfies the predicate associated to the description.

- Rule THREC says that a record is characterized by the values of its components.
- 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 $\mathsf{quo}_{T/q}$ 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 $\mathsf{quo}_{T/q}$ maps each value of the quotiented type to its equivalence class, which is a value of the quotient type.
- Rules THPROJSUB and THEMBSUB say that projectors/embedders for record/sum subtypes agree with projectors/embedders for record/sum supertypes.
- Rule THSUB says that every value of a subtype satisfies the predicate that characterizes the subtype with respect to a supertype.

3.8 Proofs

The previous subsections have defined judgements of the forms

$$\begin{array}{l} \vdash cx : \text{CONTEXT} \\ \vdash sp : \text{SPEC} \\ cx \vdash T : \text{TYPE} \\ cx \vdash T_1 \approx T_2 \\ cx \vdash T_1 \prec_r T_2 \\ cx \vdash e : T \\ cx \vdash e \end{array}$$

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 $cx \vdash T : \text{TYPE}$ 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.

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 judgements 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].

6 Other Metaslang constructs

6.1 Record updates

A record update as defined in [1] is just an abbreviation for an explicit record construction whose fields are assigned projections from the two expressions, as appropriate.

A record updater has the form

$$\ll_{\prod_i f_i\ T_i, \prod_i f'_i\ T'_i, \prod_i f''_i\ T''_i}$$

where $\prod_i f_i\ T_i, \prod_i f'_i\ T'_i, \prod_i f''_i\ T''_i \in \mathit{Type}$ and $\overline{f}' \cap \overline{f}'' = \emptyset$. Its type is

$$(\prod_i f_i\ T_i \times \prod_i f'_i\ T'_i) \rightarrow (\prod_i f_i\ T_i \times \prod_i f''_i\ T''_i) \rightarrow (\prod_i f_i\ T_i \times \prod_i f'_i\ T'_i \times \prod_i f''_i\ T''_i)$$

i.e. it operates on two records whose common fields have the same types and returns a record with the union of all the fields. The resulting record is obtained by putting together the second record with the fields of the first one that do not appear in the second record. In other words, we start with the first record, overwrite the common fields with those from the second record, and add the extra fields from the second record. This is defined by the abbreviation

$$\ll \quad \longrightarrow \quad \lambda\gamma_1 : T'.\ \lambda\gamma_2 : T''.\ \langle \overline{f} \leftarrow (\gamma_2.\overline{f})\ \overline{f}' \leftarrow (\gamma_1.\overline{f}')\ \overline{f}'' \leftarrow (\gamma_2.\overline{f}'') \rangle$$

where we leave the record types that tag $\ll$ implicit, where T' and T'' respectively stand for $\prod_i f_i\ T_i \times \prod_i f'_i\ T'_i$ and $\prod_i f_i\ T_i \times \prod_i f''_i\ T''_i$, and where the notation $\overline{f} \leftarrow e.\overline{f}$ stands for $f_1 \leftarrow e.f_1\ \ldots\ f_n \leftarrow e.f_n$. Recall that γ_i are distinct fixed but unspecified names in $\mathcal{N}$, as previously explained.

The infix form is defined by the abbreviation

$$e_1 \ll e_2 \quad \longrightarrow \quad \ll\ e_1\ e_2$$

where again we leave the tagging record types implicit.

6.2 Simple let expressions

A simple let expression has the form

$$\mathsf{let}\ v : T \leftarrow e\ \mathsf{in}\ e'$$

where $v \in \mathcal{N}$, $T \in \mathit{Type}$, and $e, e' \in \mathit{Exp}$.

We introduce the abbreviation

$$\mathsf{let}\ v : T \leftarrow e\ \mathsf{in}\ e' \quad \longrightarrow \quad (\lambda v : T.\ e')\ e$$

which captures the class of (non-recursive) let expressions defined in [1] such that the pattern is just a variable. Let expressions with more complex patterns are defined later.

6.3 Binding conditionals

A binding conditional, currently absent from [1], has the form

$$\begin{aligned}\mathsf{cond}_T\ \langle & v_{1,1}\!:\!T_{1,1},\ldots,v_{m_1,1}\!:\!T_{m_1,1}.\ b_1\rightarrow e_1\\ & \ldots\\ & v_{1,n}\!:\!T_{1,n},\ldots,v_{m_n,n}\!:\!T_{m_n,n}.\ b_n\rightarrow e_n\rangle\end{aligned}$$

where $T\in \mathit{Type}$, $\overline{\overline{v}}\in(\mathcal{N}^{(*)})^+$, $\overline{\overline{T}}\in(\mathit{Type}^*)^+$, and $\overline{b},\overline{e}\in \mathit{Exp}^+$.

Its intuitive meaning is the following. Each b_i is a boolean expression: if b_1 holds, the result of the conditional is e_1; otherwise, if b_2 holds, the result is e_2; and so on. At least one b_i must hold. Each branch i binds zero or more variables $\overline{v}_i$, whose scope is not only the condition b_i, but also the result expression e_i. Each branch i reads as: if there exist $\overline{v}_i$ with respective types $\overline{T}_i$ such that b_i holds, then the result is e_i, which can refer to the bound variables. The value of e_i must be the same for all values assigned to $\overline{v}_i$ that make b_i true. All the e_i must have type T.

We introduce the abbreviations

$$\begin{array}{rcl}\mathsf{cond}_T\ \langle\overline{v}\!:\!\overline{T}.\ b\rightarrow e\rangle & \longrightarrow & \iota\gamma_{\overline{v},b,e}\!:\!T.\exists\overline{v}\!:\!\overline{T}.\ (b\wedge\gamma_{\overline{v},b,e}\equiv e)\\ \mathsf{cond}_T\ \langle\overline{v}_1\!:\!\overline{T}_1.\ b_1\rightarrow e_1 & \longrightarrow & \mathsf{if}\ (\exists\overline{v}_1\!:\!\overline{T}_1.\ b_1)\\ \ldots & & \quad(\iota\gamma_{\overline{v}_1,b_1,e_1}\!:\!T.\exists\overline{v}_1\!:\!\overline{T}_1.\ (b_1\wedge\gamma_{\overline{v}_1,b_1,e_1}\equiv e_1))\\ \overline{v}_n\!:\!\overline{T}_n.\ b_n\rightarrow e_n\rangle & & \quad(\mathsf{cond}_T\ \langle\overline{v}_2\!:\!\overline{T}_2.\ b_2\rightarrow e_2\ \ldots\ \overline{v}_n\!:\!\overline{T}_n.\ b_n\rightarrow e_n\rangle)\end{array}$$

where $\gamma_{\overline{v},b,e}$ is, for each triple $\langle\overline{v},b,e\rangle\in\mathcal{N}^*\times \mathit{Exp}\times \mathit{Exp}$, a fixed but unspecified name in $\mathcal{N}$ such that $\gamma_{\overline{v},b,e}\notin\overline{v}\cup\mathcal{FV}(b)\cup\mathcal{FV}(e)$ and where the second abbreviation only applies when $n>1$.

6.4 Pattern matching

A pattern matching expression, currently present in [1] in a more limited form than defined here, has the form

$$\begin{aligned}\mathsf{case}_{T,T'}\ e\ \langle & v_{1,1}\!:\!T_{1,1},\ldots,v_{m_1,1}\!:\!T_{m_1,1}.\ p_1\rightarrow e_1\\ & \ldots\\ & v_{1,n}\!:\!T_{1,n},\ldots,v_{m_n,n}\!:\!T_{m_n,n}.\ p_n\rightarrow e_n\rangle\end{aligned}$$

where $T,T'\in \mathit{Type}$, $e\in \mathit{Exp}$, $\overline{\overline{v}}\in(\mathcal{N}^{(*)})^+$, $\overline{\overline{T}}\in(\mathit{Type}^*)^+$, and $\overline{p},\overline{e}\in \mathit{Exp}^+$.

Its intuitive meaning is the following. Each p_i is a pattern expression of type T against which e, which must also have type T, is compared: if e matches p_1, the result of the case expression is e_1; otherwise, if e matches p_2 holds, the result is e_2; and so on. The target expression e must match at least one p_i. Each branch i binds zero or more variables $\overline{v}_i$, whose scope is not only the pattern p_i, but also the result expression e_i. Here, "e matches p_i" means that $e\equiv p_i$ for some values of $\overline{v}_i$ of types $\overline{T}_i$. Every branch i reads as: if there exist $\overline{v}_i$ with respective types $\overline{T}_i$ such that $e\equiv b_i$ holds, then the result is e_i, which can refer to the bound variables. The value of e_i must be the same for all values assigned to $\overline{v}_i$ that make $e\equiv b_i$ true. All the e_i must have type T'.

We introduce the abbreviation

$$\mathsf{case}_{T,T'}\ e\ \langle\overline{v}_i\!:\!\overline{T}_i.\ p_i\rightarrow e_i\rangle_i\quad\longrightarrow\quad \mathsf{let}\ \gamma_{\overline{\overline{v}},\overline{p},\overline{e}}\!:\!T\leftarrow e\ \mathsf{in}\ \mathsf{cond}_{T'}\ \langle\overline{v}_i\rangle_i\overline{T}_i(\gamma_{\overline{\overline{v}},\overline{p},\overline{e}}\equiv p_1)$$

where $\gamma_{\overline{\overline{v}},\overline{p},\overline{e}}$ is, for each triple $\langle\overline{\overline{v}},\overline{p},\overline{e}\rangle\in(\mathcal{N}^{(*)})^+\times \mathit{Exp}^+\times \mathit{Exp}^+$, a fixed but unspecified name in $\mathcal{N}$ such that $\gamma_{\overline{\overline{v}},\overline{p},\overline{e}}\notin\bigcup_i\overline{v}_i\cup\bigcup_i\mathcal{FV}(p_i)\cup\bigcup_i\mathcal{FV}(e_i)$. The reason why we introduce this extra $\gamma_{\overline{\overline{v}},\overline{p},\overline{e}}$ variable via a simple let expression is that we do not want the bindings of the branches to include the target expression e, which would happen if we put e directly inside the equalities in the branches.

Here, the patterns p_i can be any expressions. In [1], patterns are a separate syntactic category, which can be regarded as a subset of expressions.

6.5 Let expressions

6.5.1 Non-recursive

As in [1], a non-recursive let expression is defined as a case expression with one branch. This generalizes the simple let expressions introduced earlier; the reason why we introduced simple let expressions first is that they are used to define case expressions, which are used to define general let expressions.

A non-recursive let expression has the form

$$\mathsf{let}_{T,T'}\ \overline{v}\!:\!\overline{T}.\ p\!\leftarrow\! e\ \mathsf{in}\ e'$$

where $T, T' \in \mathit{Type}$, $\overline{v} \in \mathcal{N}^{(*)}$, $\overline{T} \in \mathit{Type}^*$, and $p, e, e' \in \mathit{Exp}$.

We introduce the abbreviation

$$\mathsf{let}_{T,T'}\ \overline{v}\!:\!\overline{T}.\ p\!\leftarrow\! e\ \mathsf{in}\ e' \quad\longrightarrow\quad \mathsf{case}_{T,T'}\ e\ \langle\overline{v}\!:\!\overline{T}.\ p\!\rightarrow\! e'\rangle$$

which captures a generalization of non-recursive let expressions as defined in [1], in the same way as the pattern matching defined here generalizes the pattern matching defined in [1].

6.5.2 Recursive

Recursive let expressions are defined in terms of non-recursive let expressions and binding conditionals.

A recursive let expression has the form

$$\mathsf{let}_T\ \langle v_1\!:\!T_1\!\leftarrow\! e_1 \ldots v_n\!:\!T_n\!\leftarrow\! e_n\rangle\ \mathsf{in}\ e$$

where $T \in \mathit{Type}$, $\overline{v} \in \mathcal{N}^{(+)}$, $\overline{T} \in \mathit{Type}^+$, $\overline{e} \in \mathit{Exp}^+$, and $e \in \mathit{Exp}$.

We introduce the abbreviation

$$\mathsf{let}_T\ \langle v_i\!:\!T_i\!\leftarrow\! e_i\rangle_i\ \mathsf{in}\ e \quad\longrightarrow\quad \mathsf{let}_{\prod_i T_i, T}\ \overline{v}\!:\!\overline{T}.\ \langle\overline{v}\rangle\!\leftarrow\!\mathsf{cond}_{\prod_i T_i}\ \langle\overline{v}\!:\!\overline{T}.\ \langle\overline{v}\rangle \equiv \langle\overline{e}\rangle\!\rightarrow\!\langle\overline{v}\rangle\rangle\ \mathsf{in}\ e$$

which captures recursive let expressions as defined in [1].

6.6 Choosers

A chooser has the form

$$\mathsf{ch}_{T/q,T'}\ e$$

where $T/q, T' \in \mathit{Type}$ and $e \in \mathit{Exp}$.

We introduce the abbreviation

$$\mathsf{ch}_{T/q,T'}\ e \quad\longrightarrow\quad \lambda\gamma\!:\!T/q.\ (\mathsf{let}_{T/q,T'}\ \gamma'\!:\!T.\ \mathsf{quo}\ \gamma'\!\leftarrow\!\gamma\ \mathsf{in}\ e\ \gamma')$$

Recall that γ and γ' are distinct fixed but unspecified names in $\mathcal{N}$, as previously explained.

6.7 Embedding tests

An embedding test has the form

$$\mathsf{emb?}_{\sum_i c_i\ T_i}\ c_j$$

where $\sum_i c_i\ T_i \in \mathit{Type}$.

We introduce the abbreviation

$$\mathsf{emb?}_{\sum_i c_i\ T_i}\ c_j \quad\longrightarrow\quad \lambda\gamma\!:\!\textstyle\sum_i c_i\ T_i.\ \exists\gamma'\!:\!T_j.\ \gamma \equiv \mathsf{emb}\ c_j\ \gamma'$$

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.