Chapter 2

Generic core system

2.1 Introduction

To start our quest, we first construct an extension of ML which supports *generic values* — polymorphic values whose semantics are dependent on their evaluation type contexts, types at evaluation. Generic values cannot be achieved without run time types, since evaluation type contexts are dynamic: they may be different from the types statically derived at type checking.

We formalize core GML, our target language which is an extension of the ML language with type-dependent semantics. Additional to normal parametric polymorphic values of ML, GML supports generic polymorphic values whose semantics are type-dependent. We focus in this chapter mainly on the type aware semantics of GML. We do not introduce any user-defined generic value definition here. Generic values are considered to be *generic primitives* or their *derived generic values* only. Semantics of a generic primitive is given explicitly as a function which takes an evaluation context type argument and returns some semantic value. For example the following `plus` generic primitive provides addition functions for various data types:

```
# plus 1 2;;
- : int = 3
# plus 1.2 3.4;;
- : float = 4.6
...
```

Semantics of each data type is not defined in the language level, but given explicitly. *Derived generic values* are polymorphic values whose genericity is derived from internal use of other generic primitives and derived values in definitions. The following `double` is such a derived generic value:

```
let double = fun x → plus x x;;

# double 1;;
- : int = 2
# double 1.2;;
- : float = 2.4
```

We first define our type algebra in Section 2.2. We enlarge our types with a set of *dynamic type variables* in order to indicate type-dependent semantics of values annotated with them. Type schemes of generic values are quantified by one or more dynamic type variables. This distinction between generic and other polymorphic values in types isolates semantic rules and compilation of generics from those for normal values. We therefore easily keep the cleanness and efficiency of the non-generic part of our language as standard ML theory and implementation.

We define the syntax of the language in Section 2.3. It has the same syntactic constructs as standard ML augmented with generic primitives. To define the type-dependent semantics system, a fully type-annotated syntax of GML is introduced. Type substitution is extended to mappings over the fully type-annotated expressions which may modify these annotations.

The evaluation of GML is defined in Section 2.4 as an operational semantic system. Unlike other usual ML semantic formalizations without types, the semantics must be defined over fully type-annotated GML expressions, since semantics of generic values are type-dependent. During evaluation of an instance of a polymorphic identifier, the type instantiation of this identifier is applied to its semantic value. Thanks to this dynamic *type context instantiation* in semantics, generic instances are evaluated under their correct type contexts. The semantics of a generic primitive is explicitly given as a function which takes its evaluation type context as an argument and returns the corresponding semantic value in this context. For example, let $\texttt{ident:}\forall\delta\texttt{.}\delta$ be a generic primitive which returns the identity element of its type context (`1` for integer, the identity function for $\alpha \rightarrow \alpha$ and so on). Then its evaluation can be expressed by an explicitly given function from types to values:

$$\texttt{ident : int} \mapsto \texttt{1}$$
$$\texttt{ident : string} \mapsto \texttt{""}$$

A definition of a derived generic value is stored in the evaluation environment as a *generic closure*. A generic closure is abstracted by a set of type variables and is executed with some type context for these abstracted type variables, where an instance of the corresponding derived generic value is evaluated. The following derived generic value definition

```
let f = ident;;
```

stores a generic closure $gclos(E, \_, \{\delta\}, \texttt{ident} : \delta)$ for instance. When an instance of `f` is evaluated under type τ, it executes the code $\texttt{ident:}\delta$ applying a type instantiation $(\delta \Rightarrow \tau)$:

$$(\delta \Rightarrow \tau)(\texttt{ident} : \delta) = \texttt{ident} : \tau \ \mapsto \ \text{the identity of type } \tau$$

We finally build a type system for GML that prevents run time type error during evaluation in Section 2.5. The GML type system is a variant of the ML type system and most rules are straightforward, but its type judgments must be defined against fully type-annotated expressions, since the semantics suppose fully type annotations, too. The soundness of the type system is proven: once a GML program is validated by the type system, its execution is free from potential run time type errors. We also define a type reconstruction algorithm in Section 2.6 which is complete with respect to the typing rules.

2.2 Types and schemes

2.2.1 Types and their tools

Before defining the syntax of our language, we define the types, since the syntax will incorporate them. Distinction between normal and generic polymorphic values in the type system is essential to our formalization. Special semantic rules and compilation algorithms will be applied only to expressions with dynamic types. On the other hand, normal, non generic expressions are treated as usual, so that their semantic simplicity and compilation efficiency can be maintained.

Two kinds of sets of type variables are available in our type system. The set $\mathbb{R}$ contains *regular type variables* written α, β ..., which are familiar in ML languages. Another set $\mathbb{D}$ contains *dynamic type variables* δ, δ' ... to denote that the polymorphism with these variables is *generic*, non parametric polymorphic. To range on both kinds of type variables we use ν. We also have some basic types *base*, typically `unit` or `int` As in other ML families, we have function types and tuples. Types are written $\tau, \tau_1, \ldots$ and type schemes $\sigma, \sigma_1, \ldots$.

$$
\begin{array}{rcll}
\tau & ::= & base & \text{Basic types} \\
 & \mid & \alpha & \text{Regular type variables} \\
 & \mid & \delta & \text{Dynamic type variables} \\
 & \mid & \tau \to \tau & \text{Functional types} \\
 & \mid & \tau * \tau & \text{Tuple types} \\
 & & & \\
\sigma & ::= & \forall \nu \ldots \nu . \tau & \text{Type schemes}
\end{array}
$$

The definition of a type scheme is standard, and both regular and dynamic type variables can be quantified. Instead of listing quantified variables like $\forall \nu_1 \ldots \nu_n . \tau$, we sometimes write them as a set as in $\forall V . \tau$ where $V = \{\nu_1, \ldots, \nu_n\}$. Usually a type scheme $\forall \emptyset . \tau$ is written simply τ.

Free and bound variables

$FV(\tau)$ is the set of *free type variables* of τ and is defined inductively on the structure of the type expressions. Any type variable in a type expression being free, the free type variables of a type coincides with the set of type variables appearing in the type:

$$
\begin{array}{rclcrcl}
FV(base) & = & \emptyset & & & & \\
FV(\alpha) & = & \{\alpha\} & & FV(\delta) & = & \{\delta\} \\
FV(\tau \to \tau') & = & FV(\tau) \cup FV(\tau') & & FV(\tau * \tau') & = & FV(\tau) \cup FV(\tau')
\end{array}
$$

Because we have two kinds of variables, it will be convenient to introduce some more notations. $RV(\tau)$ and $DV(\tau)$ are the set of regular and dynamic free variables occurring in τ respectively:

$$
RV(\tau) = FV(\tau) \cap \mathbb{R} \qquad DV(\tau) = FV(\tau) \cap \mathbb{D}
$$

Bound variables of a type scheme σ, $BV(\sigma)$, are the type variables quantified in σ:

$$BV(\forall\nu_1 \dots \nu_n.\tau) = \{\nu_1, \dots, \nu_n\}$$

Like RV and DV, we define BRV and BDV, *bound regular variables* and *bound dynamic variables*:

$$BRV(\tau) = BV(\sigma) \cap \mathbb{R} \qquad BDV(\tau) = BV(\sigma) \cap \mathbb{D}$$

Then the set of *free variables* of a type scheme σ, $FV(\sigma)$, is defined as the rest, the set of non-bound variables.

$$FV(\forall\nu_1 \dots \nu_n.\tau) = FV(\tau) \setminus BV(\tau)$$

We say a type variable ν is *free* in a type scheme σ if ν is in $FV(\sigma)$. If a type scheme has no free type variables, we call it *closed*. We also naturally extend the notion of RV and DV to the type schemes:

$$RV(\sigma) = FV(\sigma) \cap \mathbb{R} \qquad DV(\sigma) = FV(\sigma) \cap \mathbb{D}$$

Finite mapping and environments

Before we introduce various type manipulations, we define the notion and notations of *finite mapping* and *environment*.

A mapping from a set P to Q is finite if its *definition* is given as a finite set of *bindings* of pair of elements of P and Q. If for a finite number of elements of P, say $p_1, \dots, p_n$, a finite mapping F maps p_i to q_i, i.e. $\forall 1 \leq i \leq n,\ F(p_i) = q_i \in Q$, then the definition of F is given as:

$$F \equiv \{p_1 \Rightarrow q_1, \dots, p_n \Rightarrow q_n\}$$

We regard a finite mapping and its definition as identical. Of course F cannot contain more than one binding for some p_i. Bindings $\{p_1 \Rightarrow q_1, \dots, p_n \Rightarrow q_n\}$ is often simply written as $\{p_i \Rightarrow q_i\}$, when the range of i, $1 \leq i \leq n$ is clear from the context.

The *domain* of a finite mapping F, $Dom(F)$ is the (finite) subset of P where $F(p)$ is defined:

$$Dom(F) = \{p_i \in P \mid \exists q_i, (p_i \Rightarrow q_i) \in F\}$$

For two finite mappings F_1 and F_2 from P to Q whose domains are disjoint, e.g. $Dom(F_1) \cap Dom(F_2) = \emptyset$, we can define their *direct sum*, $F_1 + F_2$:

$$F_1 + F_2 = F_1 \cup F_2$$

The condition on their domains is required for the well-formedness of the sum.

Overriding bindings of a finite map is also useful. We write $F_1\{F_2\}$ for a finite mapping F_1 overridden by another finite mapping F_2:

$$F_1\{F_2\} = \{(p{\Rightarrow}F(p)) \mid p \in \mathit{Dom}(F_1) \text{ and } p \notin \mathit{Dom}(F_2)\} + F_2$$

In $F_1\{F_2\}$, the F_2's bindings $(p{\Rightarrow}q)$ override those of F_1 if $p \in \mathit{Dom}(F_1)$, otherwise just add themselves to F_1. If $F_2 = \{p_i{\Rightarrow}q_i\}$, we also use the following notation

$$F_1\{p_i{\Rightarrow}q_i\}$$

for $F_1\{F_2\}$. This notation emphasizes that the bindings of $p_i{\Rightarrow}q_i$ belong to the whole mapping $F_1\{F_2\}$. For example,

$$F\{p{\Rightarrow}q\}$$

is a finite mapping with a binding from p to q.

An *environment* is a finite mapping from *identifiers*, (variables in the source language), to some values, for example, type schemes, semantic values and so on. Sometimes we will use different symbols for environment bindings instead of the original '$\Rightarrow$'. For instance, a *typing environment*, a mapping from identifiers to type schemes, is written $E\{\mathtt{x} : \sigma\}$.

2.2.2 Admissible type substitutions

Type substitutions are defined as usual: a *type substitution* (or just a substitution) S is a finite mapping from type variables (either regular or dynamic) to type expressions. A substitution S extends naturally to a homomorphism of type expressions:

$$\begin{array}{rcl}
S(\mathit{base}) & = & \mathit{base} \\
S(\nu) & = & \tau \quad \text{if } (\nu{\Rightarrow}\tau) \in S \\
S(\nu) & = & \nu \quad \text{otherwise} \\
S(\tau_1 \to \tau_2) & = & S(\tau_1) \to S(\tau_2) \\
S(\tau_1 * \tau_2) & = & S(\tau_1) * S(\tau_2)
\end{array}$$

We often identify a finite mapping from variables to types, and the homomorphism of type expressions of the same type substitution.

Definition 1 (Compositions of substitutions) *Let S_1 and S_2 be substitutions. The composition of the substitutions S_1 and S_2, $S_1 \circ S_2$ is the substitution whose definition is:*

$$S_1 \circ S_2 = \{(\nu{\Rightarrow}S_1(\tau)) \mid (\nu{\Rightarrow}\tau) \in S_2\} \cup \{(\nu{\Rightarrow}\tau) \in S_1 \mid \nu \notin \mathit{Dom}(S_2)\}$$

Theorem 1 *Let S_1 and S_2 be substitutions. For any type expression τ, $(S_1 \circ S_2)(\tau) = S_1(S_2(\tau))$.*

Proof: We let

$$S_3 = S_1 \circ S_2 = \{(\nu \Rightarrow S_1(\tau)) \mid (\nu \Rightarrow \tau) \in S_2\} \cup \{(\nu \Rightarrow \tau) \in S_1 \mid \nu \notin Dom(S_2)\}$$

S_3 is a well-formed substitution, because $Dom(\{(\nu \Rightarrow S_1(\tau)) \mid (\nu \Rightarrow \tau) \in S_2) = Dom(S_2)$ and $Dom(\{(\nu \Rightarrow \tau) \in S_1 \mid \nu \notin Dom(S_2)\}) = Dom(S_1) \setminus Dom(S_2)$ are disjoint each other. We prove $S_3(\tau) = S_1(S_2(\tau))$ for any type expression by induction on the structure of types.

Base types $S_3(base) = S_1(S_2(base)) = base$ is clear from the definition of the substitution.

Type variables If $\nu \in Dom(S_2)$, $S_3(\nu) = S_1(S_2(\nu)) = (S_1 \circ S_2)(\nu)$. If $\nu \in Dom(S_1) \setminus Dom(S_2)$, $S_3(\nu) = S_1(\nu)$. $S_1(S_2(\nu)) = S_1(\nu)$ also, because $\nu \notin Dom(S_2)$. Otherwise, $\nu \notin Dom(S_1)$ and $\nu \notin Dom(S_2)$, $S_3(\nu) = S_1(S_2(\nu)) = \nu$.

Functional types and tuples Use the induction hypothesis and the definition of S_3.

□

Lemma 1 (Substitutions and free variables) *For any type expression τ and substitution S, we have:*

$$FV(S(\tau)) = \bigcup_{\nu \in FV(\tau)} FV(S(\nu))$$

Proof: Proven by induction on the structure of τ. □

To prevent mixing regular and dynamic type variables in type substitutions, we need to restrict substitutions to *admissible substitutions*. Admissible substitutions never map dynamic type variables to types with regular type variable. For example, $\{\alpha \Rightarrow \delta',\ \delta \Rightarrow \delta'' \rightarrow \delta''\}$ is admissible but $\{\delta \Rightarrow \alpha \rightarrow \alpha\}$ is not. More precisely:

Definition 2 (Admissible substitutions)
A substitution S is admissible if

$$\forall \delta \in Dom(S),\ FV(S(\delta)) \subset \mathbb{D}$$

In our system all the substitutions must be admissible, since dynamic type variables are signs of genericity and we do not want to loose them by type substitutions. We usually omit the word "admissible", unless special care must be taken.

Theorem 2 *Compositions of substitutions preserve admissibility i.e. if S_1 and S_2 are admissible substitutions, then $S_1 \circ S_2$ is admissible.*

Proof: From Theorem 1, $S_1 \circ S_2$ maps a dynamic variable δ to $S_1(S_2(\delta))$. The admissibility of S_1 and S_2 assures that both mappings never introduce regular type variables in $S_1(S_2(\delta))$. $\square$

Definition 3 (Restriction of substitution) *Let S be a substitution and V be a set of type variables. Then the* restriction of S to V, $S|_V$ *is a substitution given by the following definition:*

$$S|_V = \{(\nu \Rightarrow \tau) \in S \mid \nu \in V\}$$

The restriction of a substitution contracts the domain of the substitution to a given set of type variables.

Lemma 2 (Restrictions preserve admissibility) *The restriction of an admissible substitution is admissible: let S be an admissible substitution and V be a set of type variables, then $S|_V$ is admissible.*

Proof: Clear from the definition of $S|_V$. $\square$

We will often use the property that a substitution S does not affect a set of type variables V, e.g. that V is distinct from both domain and codomain of S. In such case we say that V is *out of the reach of* S. (Or simply, V *is out of* S.):

Definition 4 *A set of type variables V is* out of the reach of *a substitution S, when*

$$Dom(S) \cap V = \emptyset \quad \textit{and} \quad \forall \nu \in Dom(S),\ FV(S(\nu)) \cap V = \emptyset$$

Substitution over type schemes

Substitutions can be extended to type schemes also. Before extending it, we need to define α-*equivalence* between type schemes. Two type schemes are α-equivalent if one can be obtained from the other by:

- renaming its bound variables, or
- adding or removing bound variables which do not occur in the body of the scheme.

The renamings must be admissible: they must not change the regular and dynamic attributes of type variables. The formal definition of this equivalence is given by the following two equations:

$$\forall \nu_1 \ldots \nu_n.\tau \stackrel{\alpha}{=} \forall \nu'_1 \ldots \nu'_n.S(\tau) \text{ where } S = \{\nu_i \Rightarrow \nu'_i\} \text{ and } \nu'_i \notin FV(\tau)$$
$$\forall \nu_1 \ldots \nu_n.\tau \stackrel{\alpha}{=} \forall \nu \nu_1 \ldots \nu_n.\tau \text{ where } \nu \notin FV(\tau)$$

Here, in S, all ν_i's must be distinct each other and not members of $FV(\tau)$. For example,

$$\begin{array}{l}
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{=} \forall\alpha_4\alpha_5\delta_6.\,\alpha_5 \to \delta_6 \to \alpha_3 \\
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{\neq} \forall\alpha_4\delta_5\delta_6.\,\delta_5 \to \delta_6 \to \alpha_3 \\
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{\neq} \forall\alpha_4\alpha_3\delta_6.\,\alpha_3 \to \delta_6 \to \alpha_3
\end{array}$$

The type schemes in the second line are not equivalent because the renaming between α_1 and δ_5 involves the two different kinds of type variables and therefore is not admissible. The third line is also incorrect, because the free occurrence of α_3 in the left hand side is no longer free in the right hand side.

If two type schemes are α-equivalent, we usually identify them and just write $\sigma = \sigma'$, unless subtle handling of bound variables is required. Also, if we write a type scheme $\forall\nu_1\ldots\nu_n.\tau$, we suppose that it must be in one of the minimized form: all the bound variables ν_i occur in the body τ. The tool functions BV, BRV, BDV also follow this assumption:

$$\begin{array}{l}
BV(\forall\nu_1\ldots\nu_n.\tau) = \{\nu_1,\ldots,\nu_n\} \cap FV(\tau) \\
BRV(\sigma) = BV(\sigma) \cap \mathbb{R} \\
BDV(\sigma) = BV(\sigma) \cap \mathbb{D}
\end{array}$$

Using α-equivalence, substitutions can be extended to type schemes:

$$S(\forall\nu_1\ldots\nu_n.\tau) = \forall\nu_1\ldots\nu_n.S(\tau) \quad \text{when } \{\nu_i\} \text{ is out of the reach of } S$$

We rename bound variables and do α-conversions if needed, so that $Dom(S)$ and $S(\tau)$ can contain none of the bound variables, $\nu_1\ldots\nu_n$. For example,

$$\begin{array}{rcl}
\{\alpha_1{\Rightarrow}\alpha_2, \delta_3{\Rightarrow}\delta_4\}\,(\forall\alpha_1\delta_4.\alpha_1 \to \delta_3) & \overset{\alpha}{=} & \{\alpha_1{\Rightarrow}\alpha_2, \delta_3{\Rightarrow}\delta_4\}(\forall\alpha_5\delta_6.\alpha_5 \to \delta_3) \\
& = & \forall\alpha_5\delta_6.\alpha_5 \to \delta_4
\end{array}$$

2.2.3 Instance of type schemes

Instances of type schemes are also defined as usual, but substitution are restricted to admissible ones.

Definition 5 (Instances) *A type expression τ is an* instance *of a type scheme $\sigma = \forall\nu_1\ldots\nu_n.\tau'$ and we write $\tau \le \sigma$, if there is an admissible substitution S such that*

$$Dom(S) = \{\nu_i\} \quad \textit{and} \quad \tau = S(\tau')$$

We also write $\tau \le_S \sigma$, if $\tau \le \sigma$ holds with a substitution S which suffices the above conditions. For example, $\alpha \to \delta \le_S \forall\alpha\beta.\alpha \to \beta$ where $S = \{\alpha{\Rightarrow}\alpha,\ \beta{\Rightarrow}\delta\}$. The substitution S is called the *instantiation* of σ to τ. Note that an instantiation of σ is only valid for σ itself and not for the other α-congruents of σ.

Lemma 3 (α-conversions preserve the instance relationship) *If $\tau \leq \sigma$, then for any σ' such that $\sigma' \stackrel{\alpha}{=} \sigma$, we have $\tau \leq \sigma'$ also.*

Proof: Let $\sigma = \forall \nu_1 \ldots \nu_n.\tau'$, $\sigma' = \forall \nu'_1 \ldots \nu'_m.\tau''$ and S be a renaming substitution $S = \{\nu_i \Rightarrow \nu'_j\}$ such that $S(\tau') = \tau''$. Because S is just a renaming of variables and one-to-one mapping, therefore the inverse of S, $S^{-1} = \{\nu'_j \Rightarrow \nu_i\}$ exists.

Suppose that $\tau \leq_{S'} \sigma$. Then $S'(\tau') = \tau$. From $S^{-1}(\tau'') = \tau'$, $(S' \circ S^{-1})(\tau'') = S'(\tau') = \tau$, this leads to $\tau'' \leq_{(S' \circ S^{-1})} \sigma$. $\square$

Lemma 4 (Substitution lemma of instances) *Substitutions preserves the instance relationship: if $\tau \leq_{S_0} \sigma$, for any admissible substitution S, we have $S(\tau) \leq_{S_1} S(\sigma)$, with $S_1 = (S \circ S_0)|_{BV(\sigma)}$.*

Proof: Let $\sigma = \forall V.\tau'$. Then $S_0(\tau') = \tau$. Let $S' = S_{|Dom(S)\setminus V}$, then $S(\sigma) = \forall V.S'(\tau')$.

We prove $S_1 \circ S' = S \circ S_0$. Let $\nu \notin V$, then $(S \circ S_0)(\nu) = S(\nu)$, on the other hand $(S_1 \circ S')(\nu) = S_1 \circ S(\nu)$, because $Dom(S_0) = Dom(S_1) = \{\nu_i\}$.

Otherwise, if $\nu = \nu_i$, one of the bound variables, from the definition of S_1 and $S(\nu_i) = \nu_i$, we have $(S \circ S_0)(\nu_i) = S_1(\nu_i) = (S_1 \circ S)(\nu_i)$. Therefore,

$$S_1 \circ S' = S \circ S_0, \quad (S_1 \circ S')(\tau') = (S \circ S_0)(\tau') = S(\tau)$$

Because $S(\sigma) = \forall \nu_1 \ldots \nu_n.S'(\tau')$,

$$S(\tau) \leq_{S_1} S(\sigma)$$

$\square$

Using these instances of type schemes, we can define an ordering between type schemes:

Definition 6 (More/less general type schemes) *Let σ and σ' be type schemes. We say σ is more general than σ' (or σ' is less general than σ) and write $\sigma' \leq \sigma$, if and only if $\forall \tau \leq \sigma', \tau \leq \sigma$.*

2.3 The core language and type-annotated syntax

The language GML is the core language which we use in the following chapters. It is a variant of the ML core language, a polymorphic language including the usual call-by-value λ-calculus, extended with some data constructors, tuples, basic constants, let polymorphism, and finally, generic primitives.

Here, generic values are treated as primitives or their derived values. Definitions of generic primitives are omitted, being known by their names and type schemes. For example we have $\texttt{print}_{\forall \delta.\delta \rightarrow \texttt{unit}}$ and $\texttt{plus}_{\forall \delta.\delta \rightarrow \delta \rightarrow \delta}$ as generic primitives in this chapter. In the following `g` ranges over anonymous identifiers for generic primitives. The problem of user-defined generic values is treated in in Chapter 7.

$$
\begin{array}{rcll}
e & ::= & c_\sigma \quad \text{where } FV(\sigma) = \emptyset, BDV(\sigma) = \emptyset & \text{Constants} \\
& | & \mathtt{g}_\sigma \quad \text{where } FV(\sigma) = \emptyset, BDV(\sigma) \neq \emptyset, BRV(\sigma) = \emptyset & \text{Generic primitives} \\
& | & \mathtt{x} & \text{Variables} \\
& | & \mathtt{fun}\ \mathtt{x} \rightarrow e & \lambda \text{ abstractions} \\
& | & e\ e & \text{Applications} \\
& | & e, e & \text{Tuples} \\
& | & \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e & \text{Let bindings} \\
& | & \mathtt{let\ rec}\ \mathtt{x} = e\ \mathtt{in}\ e & \text{Recursive let bindings}
\end{array}
$$

The syntax contains types: constants and generic primitives are explicitly annotated with their *closed* type schemes. Constants are non-generic: their type schemes do not contain dynamic variables. On the contrary, type schemes of generic primitives are fully generic: they are quantified by non-empty sets of dynamic type variables. We do not consider primitives with type schemes having a mix of regular and dynamic type variables. Other expression constructs, such as functions, applications, tuples, let bindings are regular. For simplicity multiple function abstractions are often abbreviated as follows:

$$\mathtt{fun}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n \rightarrow e \ \equiv\ \mathtt{fun}\ \mathtt{x}_1 \rightarrow\ \ldots \rightarrow \mathtt{fun}\ \mathtt{x}_n \rightarrow e$$

It is impossible to define the semantics of GML solely on the above syntax, because semantics of generic primitives depend on their typing. Therefore, the semantics must be defined over *fully type-annotated* GML, where all the sub-expressions have type annotations. We sometimes refer the original GML without type annotations as *bare* GML, to distinguish it from this type-annotated syntax. The grammar of fully type-annotated GML expressions e_t is defined as follows:

$$
\begin{array}{rcl}
e_t & ::= & c_\sigma : \tau \\
& | & \mathtt{g}_\sigma : \tau \\
& | & \mathtt{x}_\sigma : \tau \\
& | & \mathtt{fun}\ \mathtt{x} \rightarrow e_t : \tau \\
& | & e_t\ e_t : \tau \\
& | & e_t, e_t : \tau \\
& | & \mathtt{let}\ \mathtt{x}_{\{\nu\ldots\nu\}} = e_t\ \mathtt{in}\ e_t : \tau \\
& | & \mathtt{let\ rec}\ \mathtt{x}_{\{\nu\ldots\nu\}} = e_t\ \mathtt{in}\ e_t : \tau
\end{array}
$$

Sub-expressions have type annotations after the symbol '`:`'. Constants, generic primitives and variables now have two type informations: a type scheme σ attached to a value or primitive itself and a type expression τ which denotes how each occurrence is instantiated. Let bindings have additional type information, *generalized type variable sets* to express which type variables they generalize. For instance, in the following type-annotated expression:

$$\mathtt{let}\ \mathtt{x}_{\{\delta\}} = \mathtt{print}_{\forall\delta.\delta \rightarrow \mathtt{unit}} : \delta \rightarrow \mathtt{unit}\ \mathtt{in}\ e_t : \tau$$

`x` is bound with a polymorphic type scheme $\forall\delta.\delta \rightarrow \mathtt{unit}$ inside the evaluation of e_t. We often write $e_t : \tau$, to emphasize that the out-most type annotation of e_t is τ.

Type substitutions against type annotations

We easily extend FV to the type-annotated expressions. $FV(e_t)$, the free variables of a fully type-annotated expression e_t, is defined as follows:

$$
\begin{array}{rcl}
FV(c_\sigma : \tau) & = & FV(\tau) \\
FV(\mathtt{g}_\sigma : \tau) & = & FV(\tau) \\
FV(\mathtt{x}_\sigma : \tau) & = & FV(\tau) \cup FV(\sigma) \\
FV(\mathtt{fun\ x} \rightarrow e_t : \tau) & = & FV(e_t) \cup FV(\tau) \\
FV(e_{t1}\ e_{t2} : \tau) & = & FV(e_{t1}) \cup FV(e_{t2}) \cup FV(\tau) \\
FV(e_{t1},\ e_{t2} : \tau) & = & FV(e_{t1}) \cup FV(e_{t2}) \cup FV(\tau) \\
FV(\mathtt{let\ (rec)\ x}_{\{\nu_i\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & (FV(e_{t1}) \setminus \{\nu_i\}) \cup FV(e_{t2})
\end{array}
$$

Type variables of a let definition e_{t1} are bound by the generalized type variable set $\{\nu_i\}$ of the definition, like type variables in a type scheme are bound by its $\forall$-quantified variables.

We may consider *α-conversions of fully type-annotated expressions*. The definition of α-equivalence $\stackrel{\alpha}{=}$ can be defined as a set of equations obtained by renaming variables of generalized type variable sets of `let` expressions, similarly to the definition of $\stackrel{\alpha}{=}$ for type schemes:

$$
\begin{array}{rcl}
\mathtt{let\ x}_{\{\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau & \stackrel{\alpha}{=} & \mathtt{let\ x}_{\{\nu,\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau \\
 & & \qquad \text{when } \nu \notin FV(e_{t1}) \\
\mathtt{let\ x}_{\{\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau & \stackrel{\alpha}{=} & \mathtt{let\ x}_{\{\nu_1,\ldots,\nu'_i,\ldots,\nu_n\}} \mathtt{=} e'_{t1}\ \mathtt{in}\ e_{t2} : \tau
\end{array}
$$

where $\{\nu_i \Rightarrow \nu'_i\}$ is admissible $\nu'_i \notin FV(e_{t1})$
e'_{t1} is obtained by replacing the free occurrences of ν_i of e_{t1} to ν'_i.

As for type schemes, we usually consider α-equivalent fully type-annotated expressions as identical.

Example

Suppose that we have the following fully type-annotated expression e_t:

```
(let f{α} =
   fun x →
     let h{α} =
       fun y → (yα:α) : α → α
     in
     (h∀α.α → α:α → α) (xα:α) : α
 in
 1:int) : int
```

Renaming the generalized variable α occurring in the definition of `f` to β, we obtain an α-equivalent fully type-annotated expression e'_t:

$$
\begin{array}{l}
\texttt{(let f}_{\{\beta\}}\texttt{ =} \\
\quad \texttt{fun x} \rightarrow \\
\qquad \texttt{let h}_{\{\alpha\}}\texttt{ =} \\
\qquad\quad \texttt{fun y} \rightarrow \texttt{(y}_{\alpha}\texttt{:}\alpha\texttt{) : } \alpha \rightarrow \alpha \\
\qquad \texttt{in} \\
\qquad \texttt{(h}_{\forall\alpha.\alpha \rightarrow \alpha}\texttt{:}\beta \rightarrow \beta\texttt{) (x}_{\beta}\texttt{:}\beta\texttt{) : } \beta \\
\ \texttt{in} \\
\ \texttt{1:int) : int}
\end{array}
$$

The α's inside the definition of `h` are not renamed since they are bound by the generalized type variable occurring in `h`.

□

We write $S(e_t)$ for a fully type-annotated expression e_t in which each type annotation τ or σ is replaced by $S(\tau)$ or $S(\sigma)$ respectively. The precise definition of $S(e_t)$ is quite straightforward except the case of a `let` binding:

$$
\begin{array}{rcl}
S(c_\sigma : \tau) & = & c_\sigma : S(\tau) \\
S(\mathtt{g}_\sigma : \tau) & = & \mathtt{g}_\sigma : S(\tau) \\
S(\mathtt{x}_\sigma : \tau) & = & \mathtt{x}_{S(\sigma)} : S(\tau) \\
S(\mathtt{fun\ x} \rightarrow e_t : \tau_2) & = & \mathtt{fun\ x} \rightarrow S(e_t) : S(\tau_2) \\
S(e_{t1}\ e_{t2} : \tau) & = & S(e_{t1})\ S(e_{t2}) : S(\tau) \\
S(e_{t1}, e_{t2} : \tau) & = & S(e_{t1}), S(e_{t2}) : S(\tau)
\end{array}
$$

For the `let` case, as we did for the substitutions of type schemes, we do an implicit α-conversion of the expression, so that the generalized variable set $\{\nu_i\}$ becomes out of the reach of S and is not affected by S:

$$
\begin{array}{l}
S(\mathtt{let\ (rec)\ x}_{\{\nu_i\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \\
\qquad \mathtt{let\ (rec)\ x}_{\{\nu_i\}} = S(e_{t1})\ \mathtt{in}\ S(e_{t2}) : S(\tau) \qquad \text{Rename generalized variables if necessary}
\end{array}
$$

As an auxiliary, we define here $strip(e_t)$, a function which removes all the type annotations in e_t to recover a bare GML expression:

$$
\begin{array}{rcl}
strip(c_\sigma : \tau) & = & c_\sigma \\
strip(g_\sigma : \tau) & = & g_\sigma \\
strip(\mathtt{x}_\sigma : \tau) & = & \mathtt{x} \\
strip(\mathtt{fun\ x} \rightarrow e_t : \tau_2) & = & \mathtt{fun\ x} \rightarrow strip(e_t) \\
strip(e_{t1}\ e_{t2} : \tau) & = & strip(e_{t1})\ strip(e_{t2}) \\
strip(e_{t1}, e_{t2}) & = & strip(e_{t1}), strip(e_{t2})
\end{array}
$$

$$
\begin{array}{l}
strip(\mathtt{let\ (rec)\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \\
\qquad\qquad \mathtt{let\ (rec)\ x} = strip(e_{t1})\ \mathtt{in}\ strip(e_{t2})
\end{array}
$$

2.4 Semantics

We present the semantics of fully type-annotated GML as a structured operational semantics system. ML semantics is usually formalized without types. However, GML's semantics system does need types, since generic values are type-dependent. Derived generic values are evaluated to *generic closures*: closures for generic values. A generic closure is a closure abstracted by type variables, and its code is evaluated when an *evaluation type contexts* (or just type contexts) for the corresponding generic value is given as a type instantiation of the abstracted variables.

2.4.1 Semantic values

We define the semantic domains for the GML semantics system:

r	$::=$	v	Regular values
	$\mid$	err	Errors
v	$::=$	b_σ	Basic constants
	$\mid$	v, v	Tuples
	$\mid$	$clos(E, \_, \mathtt{x}, e_t)$	Functional closures
	$\mid$	$clos(E, \mathtt{f}, \mathtt{x}, e_t)$	Functional closures (recursive)
sv	$::=$	v	Regular values
	$\mid$	$gclos(E, \_, V, e_t)$	Generic closures
	$\mid$	$gclos(E, \mathtt{x}, V, e_t)$	Generic closures (recursive)
V	$\equiv$	$\{\nu_1, \ldots, \nu_n\}$	Generalized type variables
E	$::=$	$\{\mathtt{x} = sv, \ \ldots \ \mathtt{x} = sv\}$	Environment

An evaluation of an expression returns a *result*. A result r may be a *semantic value* or an *error*. A semantic value is a successful result of an evaluation. Actually, only a subset of semantic values called *regular values* v are returned as results. The other semantic values, i.e. generic closures are never returned as results.

Regular values v consist in basic constants, tuples and functional closures. Constants are type scheme annotated and they must be closed. A *closure* is a pair of an environment and a piece of code which is abstracted by variables. It is evaluated when semantic values for the abstracted variables are supplied: the evaluation of its code is performed under its environment and the bindings of abstracted variables to the supplied values. A function $\mathtt{fun}\ \mathtt{x} \rightarrow e_t$ is evaluated to a *functional closure* $clos(E, \_, \mathtt{x}, e_t)$, and its body e_t is executed when it is applied to the argument `x`. In the case that the closure is recursively defined by a `let rec` expression, the defined identifier `f` is attached to the closure like $clos(E, \mathtt{f}, \mathtt{x}, e_t)$, so that the occurrences of `f` inside the body e_t can be evaluated to this closure itself. In the non recursive case, we write '_' instead of an identifier. We often use a meta variable symbol $*$ for a recursive identifier `f` or _.

An *evaluation environment* E is an environment that maps identifiers to *semantic values* sv. *Semantic values* sv are regular values v or *generic closures* $gclos(E, *, V, e_t)$, another kind of closures. Generic closures only appear inside evaluation environments. A generic closure $gclos(E, *, V, e_t)$ represents a derived generic value whose definition is e_t. The generic closure is abstracted by a set of *generalized type variables* V of the derived generic value, and its definition e_t is evaluated when an *evaluation type context* is available as an type instantiation of V. A variable `x` is written in a generic closure as $gclos(E, \mathtt{x}, V, e_t)$, if the value is defined recursively and bound to `x`, as in recursive functional closures. Same as type schemes and `let` type-annotated expressions, the type variables in V bind their occurrences in the type annotations of e_t. Therefore α-equivalence can be defined between generic closures:

$$\begin{array}{rcl} gclos(E, *, \{\nu_1, \ldots, \nu_n\}, e_t) & \stackrel{\alpha}{=} & gclos(E, *, \{\nu, \nu_1, \ldots, \nu_n\}, e_t) \\ & & \text{when } \nu \notin FV(e_t) \\ gclos(E, *, \{\nu_1, \ldots, \nu_n\}, e_t) & \stackrel{\alpha}{=} & gclos(E, *, \{\nu'_1, \ldots, \nu'_n\}, \{\nu_i \Rightarrow \nu'_i\}\, e_t) \\ & & \text{when } \{\nu_i \Rightarrow \nu'_i\} \text{ is admissible and } \nu'_i \notin FV(e_t) \end{array}$$

Since closures contain types, type substitutions are extended for semantic values and evaluation environments:

$$\begin{array}{rcl} S(b_\sigma) & = & b_\sigma \\ S(v_1, v_2) & = & S(v_1), S(v_2) \\ S(clos(E, *, \mathtt{x}, e_t)) & = & clos(S(E), *, \mathtt{x}, S(e_t)) \\ S(gclos(E, *, V, e_t)) & = & gclos(S(E), *, V, S(e_t)) \\ S(E) & = & \{(\mathtt{x} = S(sv)) \mid (\mathtt{x} = sv) \in E\} \end{array}$$

When we apply a substitution S to a generic closure $gclos(E, *, V, e_t)$, we may have to rename generalized type variables so that S does not affect bindings of V in e_t.

There is no special semantic value for generic primitives. Their evaluations are supposed to be always successful and their results are given explicitly depending on their evaluation type contexts.

Semantic values for primitives

For simplicity of the later arguments of compilation of GML, we assume that semantic values for constants and primitives must be restricted within the following set of *semantic values for primitives* P:

$$b_\sigma \in P$$

$$\frac{v_1 \in P \quad v_2 \in P}{v_1, v_2 \in P}$$

$$\frac{e_t \text{ has no generic primitives inside.}}{clos(\emptyset, *, \mathtt{x}, e_t) \in P}$$

The closures in P are restricted to those with the empty evaluation environment whose codes e_t contain no generic primitives.

2.4.2 evaluation rules

Our evaluation system is defined for fully type-annotated GML expressions. We use *evaluation judgments* in the form of $E \models e_t : \tau \Longrightarrow r$. The out-most type annotation τ of e_t is called the *evaluation type context* for e_t. The judgment reads that "in the evaluation environment E and under the evaluation type context τ, a fully type-annotated expression e_t is evaluated to r." When an error occurs, the semantics system gives up all the evaluation procedure and returns *err* immediately. In the following, we introduce and comment the evaluation rules one by one.

Constants and primitives

$$\textbf{(e-cnst)} \quad \frac{\tau \leq_S \sigma}{E \models c_\sigma : \tau \Longrightarrow S(\mathit{cnst}(c_\sigma))}$$

Semantics of constants are explicitly given by the function *cnst*. The returned semantic values are restricted within the set of semantic values for primitives: $\mathit{cnst}(c_\sigma) \in P$. If c_σ is a basic constant b_σ, for example an integer, then $\mathit{cnst}(c_\sigma)$ returns itself. If c_σ is a primitive non-generic function, then $\mathit{cnst}(c_\sigma)$ will return a closure $\mathit{clos}(\emptyset, *, \mathtt{x}, e_t)$, where e_t has no generic primitives occurrence. We apply the instantiation S where $\tau \leq_S \sigma$ to insure that the returned value is semantically well-typed.

$$\textbf{(e-gen)} \quad E \models \mathsf{g}_\sigma : \tau \Longrightarrow \mathit{gen}(\mathtt{g}, \tau)$$

Semantics of generic primitives are given by *gen*, the *generic semantics query function.* $\mathit{gen}(\mathtt{g}, \tau)$ returns the semantic value of generic primitive $\mathtt{g}$ in the type context τ. Like *cnst*, $\mathit{gen}(\mathtt{g}, \tau)$ must return a primitive semantic value: $\mathit{gen}(\mathtt{g}, \tau) \in P$. For example, for the generic printing function $\mathtt{print}_{\forall\delta.\delta \to \mathtt{unit}}$ evaluated under the evaluation type context $\mathtt{int} \to \mathtt{unit}$, $\mathit{gen}(\mathtt{print}, \mathtt{int} \to \mathtt{unit})$ returns a functional closure which receives an integer and prints it out. We suppose that the generic primitives are *total*: *gen* must return some semantic value different from *err* for any type context τ.

Tuples

$$\textbf{(e-tpl)} \quad \frac{E \models e_{t1} \Longrightarrow v_1 \quad E \models e_{t2} \Longrightarrow v_2}{E \models (e_{t1}, e_{t2}) \Longrightarrow v_1, v_2}$$

$$\textbf{(e-tpl-err)} \quad \frac{E \models e_{ti} \Longrightarrow err}{E \models (e_{t1}, e_{t2}) \Longrightarrow err}$$

The semantic value of a tuple is the tuple of semantic values of its contents. However, if one of the sub-expression is evaluated to *err*, then the whole evaluation also fails immediately.

Variables

The rules for variables are mainly separated into two cases, depending on the kind of environmental semantic values that bound to them. The first is for regular values, and the second is for generic closures.

$$\textbf{(e-nvar)}\quad \frac{\tau \leq_S \sigma}{E\{\mathtt{x} = v\} \models \mathtt{x}_\sigma : \tau \Longrightarrow S(v)}$$

If x is bound by a regular value v in the environment, we return v as its semantic value applied the instantiation S, where $\tau \leq_S \sigma$.

$$\textbf{(e-gvar)}\quad \frac{\tau \leq_S \forall V.\tau' \qquad E' \models S(e_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

$$\textbf{(e-gvar-r)}\quad \frac{\begin{array}{c}\exists S,\ \tau \leq_S \forall V.\tau' \\ E'\{\mathtt{f} = gclos(E', \mathtt{f}, V, e_t : \tau')\} \models S(e_t : \tau') \Longrightarrow r\end{array}}{E\{\mathtt{x} = gclos(E', \mathtt{f}, V, e_t : \tau')\} \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

$$\textbf{(e-gvar-err)}\quad \frac{\begin{array}{l}S \text{ such that } \tau \leq_S \forall V.\tau' \text{ does not exist,} \\ \text{or } \forall V.\tau' \overset{\alpha}{\not\equiv} \sigma\end{array}}{E\{\mathtt{x} = gclos(E', *, V, e_t : \tau')\} \models \mathtt{x}_\sigma : \tau \Longrightarrow err}$$

If x is bound by a generic closure, we execute the expression in the generic closure, since we have now the evaluation type context τ for it. We require τ to be an instance of the type scheme of the closure $\forall V.\tau'$. It is also necessary that the type scheme annotation σ of the variable x is α-equivalent to $\forall V.\tau'$. The instantiation S indicates how generalized variables V are actually instantiated at evaluation of the stored code e_t. We evaluate $S(e_t)$ under E' so that the generic values inside e_t can also be evaluated in their correct evaluation type context. We return the evaluation result of $S(e_t)$ as one for x. We do *not* apply the instantiation S to the environment E' in the generic closure, because we can assume that S has no relation to E' by renaming the bound variables V.

$$\textbf{(e-var-err)}\quad \frac{\mathtt{x} \notin Dom(E)}{E \models \mathtt{x}_\sigma : \tau \Longrightarrow err}$$

Absence of a binding for variable x in the evaluation environment results in an error.

Functions

$$\textbf{(e-fun)}\quad E \models \mathtt{fun\ x} \rightarrow e_t \Longrightarrow clos(E, \_, \mathtt{x}, e_t)$$

A function is evaluated to its closure, which consists of the current evaluation environment, the abstracted variable x, and the body expression of the function. The body expression is evaluated when a semantic value of its argument is given to it.

Let bindings

Evaluations of let bindings are separated into two groups, depending on whether they define derived generic values. We can distinguish them by checking the generalized type variable set attached to a let identifier contains dynamic type variables or not. In each part we have two rules, for non-recursive and recursive bindings.

First we look at non-generic let bindings. They store a binding of a variable to a semantic value.

$$\textbf{(e-let)}\ \frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} \Longrightarrow v \quad E\{\mathtt{x} = v\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V\ \mathtt{=}\ e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textbf{(e-let-err)}\ \frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} \Longrightarrow \mathit{err}}{E \models \mathtt{let}\ \mathtt{x}_V\ \mathtt{=}\ e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow \mathit{err}}$$

In a non-recursive `let`, first we evaluate the definition e_{t1}, then bound the variable `x` with its result in the environment, then evaluate the body e_{t2} under the updated environment. The result of the whole expression is equal to the result of e_{t2}. If the evaluation of e_{t1} fails, the whole expression also fails.

$$\textbf{(e-letrec)}\ \frac{\begin{array}{c} V \cap \mathbb{D} = \emptyset \\ E\{\mathtt{f} = \mathit{clos}(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} \Longrightarrow r \end{array}}{E \models \mathtt{let\ rec}\ \mathtt{f}_V\ \mathtt{=}\ \mathtt{fun}\ \mathtt{x} \rightarrow e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textbf{(e-letrec-err)}\ \frac{V \cap \mathbb{D} = \emptyset \quad \text{if } e_{t1} \text{ is non-function}}{E \models \mathtt{let\ rec}\ \mathtt{f}_V\ \mathtt{=}\ e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow \mathit{err}}$$

Recursive `let` expressions are handled differently. We require that its definition must have the form of a function. The definition is converted immediately to a recursive closure $\mathit{clos}(E, \mathtt{f}, \mathtt{x}, e_{t1})$, and the body of the function e_{t1} will be evaluated when the recursive function `f` is applied later. We add a binding of `x` with the recursive closure to the environment and evaluate the body e_{t2} in this updated environment. The bindings of non-functional recursive values are not permitted, otherwise evaluation of definitions may fall into infinite loops.

Now we move on to the generic let expressions:

$$\textbf{(e-g-let)}\ \frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = \mathit{gclos}(E, \_, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V\ \mathtt{=}\ e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textbf{(e-g-letrec)}\ \frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = \mathit{gclos}(E, \mathtt{x}, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let\ rec}\ \mathtt{x}_V\ \mathtt{=}\ e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

`let` expressions which bind generic values are differently handled from non-generic let. Their definitions are stored in evaluation environments as generic closures and are evaluated when they receive their evaluation type context by (e-gvar) and (e-gvar-r) rules. Unlike the non-generic recursive `let`, the generic recursive `let` has no such restriction that the definition must be a function. This is because its definition is always evaluated to a generic closure even if the definition is not a function. Therefore, there is no danger of infinite evaluation loop of the definition.

As we saw, the generic closures appear just in evaluation environments. They are never returned as a result of any evaluation.

Applications

$$\textbf{(e-app1)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow clos(E', \_, \mathtt{x}, e_t) \qquad E'\{\mathtt{x} = v_2\} \models e_t \Longrightarrow r}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

$$\textbf{(e-app2)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow clos(E', \mathtt{f}, \mathtt{x}, e_t) \qquad E'\{\mathtt{f} = clos(E', \mathtt{f}, \mathtt{x}, e_t),\ \mathtt{x} = v_2\} \models e_t \Longrightarrow r}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

$$\textbf{(e-app-err1)}\quad \frac{E \models e_{t2} \Longrightarrow err}{E \models e_{t1}\ e_{t2} \Longrightarrow err}$$

$$\textbf{(e-app-err2)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow r \quad \text{where } r \neq clos(\ldots)}{E \models e_{t1}\ e_{t2} \Longrightarrow err}$$

In applications, we evaluate an argument e_{t2}, then a function e_{t1}[1]. The evaluation of e_{t1} must result in a closure. We add a binding of `x` to the result of e_{t2} into the environment E' in the closure, and execute the closure code e_t in this new environment. For recursive closures, we also bind the closure itself in the new environment so that the closure can be recursively available in the evaluation of the functional body e_t. If an error occurs in the evaluation of the argument e_{t2} or the function e_{t1} is not evaluated to a closure, the application returns an error.

Example

Let $\mathtt{g}_{\forall\delta.\delta\ \to\ \mathtt{unit}}$ be a generic primitive, then the following type-annotated GML expression

$$\begin{array}{l}\mathtt{let\ f}_{\{\delta'\}}\ \mathtt{= (fun\ x} \rightarrow \mathtt{(g}_{\forall\delta.\delta\ \to\ \mathtt{unit}} : \delta' * \delta' \rightarrow \mathtt{unit)\ (x,x)} : \delta' \rightarrow \mathtt{unit)} \\ \ \mathtt{in} \\ \quad \mathtt{((f : int} \rightarrow \mathtt{unit)\ (1_{int} : int) : unit)}\end{array}$$

[1]This evaluation order is compatible with the Objective Caml implementation

is evaluated as follows (evaluation environments and type annotations are omitted if they are clear):

$$\frac{\{\mathtt{f} = gclos(\emptyset, \_, \{\delta\}, \mathtt{fun\ x} \rightarrow \mathtt{g\ (x,x)} : \delta \rightarrow \mathtt{unit})\} \models (\mathtt{f} : \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{1} \Longrightarrow r}{\emptyset \models \mathtt{let\ f = fun\ x} \rightarrow \mathtt{g\ (x,x)\ in\ f\ 1} \Longrightarrow r}$$

First we store the definition of `f` inside the environment and evaluate `f 1`. The result r of the original expression is as same as the result of `f 1`.

$$\{\mathtt{f} = \ldots\} \models \mathtt{1} \Longrightarrow \mathtt{1}$$

The argument `1` is evaluated to itself by (e-cnst).

The generic instance `f` has the type annotation $\mathtt{int} \rightarrow \mathtt{unit}$, which is an instance of the type scheme of `f`, $\mathtt{int} \rightarrow \mathtt{unit} \leq_S \forall\delta.\delta \rightarrow \mathtt{unit}$ where the instantiation $S = \{\delta \Rightarrow \mathtt{int}\}$:

$$\frac{\begin{array}{l}\emptyset \models S(\mathtt{fun\ x} \rightarrow \mathtt{g\ (x,x)} : \delta \rightarrow \mathtt{unit}) \\ \qquad\qquad \Longrightarrow clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)})\end{array}}{\begin{array}{l}\{\mathtt{f} = gclos(\emptyset, \_, \{\delta\}, \mathtt{fun\ x} \rightarrow \mathtt{g\ (x,x)} : \delta \rightarrow \mathtt{unit})\} \models \mathtt{f} \\ \qquad\qquad \Longrightarrow clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)})\end{array}}$$

The evaluation of `f` by (e-gvar) is equal to the result of the code $S(\mathtt{fun\ x} \rightarrow \mathtt{g\ (x,x)} : \delta \rightarrow \mathtt{unit})$, the definition expression of `f` whose annotations are now instantiated by S. It returns a closure $clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)})$ where `g`'s evaluation type context is instantiated to `int * int` $\rightarrow$ `unit`.

To evaluate `f 1`, the last thing we need is to store the semantic value for the argument `1` in the environment and execute the code in the closure $(\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)}$ under the updated environment by (e-app):

$$\{\mathtt{x} = \mathtt{1}\} \models (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)} \Longrightarrow r$$

Here the argument `(x,x)` is evaluated to:

$$\{\mathtt{x} = \mathtt{1}\} \models \mathtt{(x,x)} \Longrightarrow \mathtt{(1,1)}$$

And the generic instance `g` is evaluated by (e-gen):

$$\{\mathtt{x} = \mathtt{1}\} \models (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}) \Longrightarrow gen(\mathtt{g}, \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}) = clos(\emptyset, \_, \mathtt{x}, e_t)$$

Here we assume that the semantic function gen returns a proper semantic value for `g` under $\mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}$, some closure $clos(\emptyset, \_, \mathtt{x}, e_t)$ which receives a tuple of integers and returns a unit `()`:

$$\{\mathtt{x} = \mathtt{(1,1)}\} \models e_t \Longrightarrow \mathtt{()}$$

This means the above application $(\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x,x)}$ is evaluated to `()`, and therefore $r = \mathtt{()}$. □

2.5 Type system

Like usual untyped ML language, if GML programs are not verified by some typing system, they may raise run time type errors at evaluation. Such errors are hard to fix without help of static typing. We must type-check fully type-annotated GML expressions and restrict them to the *well-typed* ones so that well-typed programs can never fall into evaluation errors. Our typing system is basically as same as many systems whose origin is Damas-Milner's[DM82]. Main difference is that in ours type checking is carried out against expressions already fully type-annotated.

2.5.1 Typing environments

A *typing environment* Γ is an environment from identifiers to type schemes. We need to extend our tools for types also for typing environments. $FV(\Gamma)$ is the set of variables freely occur inside Γ's bindings:

$$FV(\Gamma) = \bigcup_{(\mathtt{x}:\sigma)\in\Gamma} FV(\sigma)$$

$S(\Gamma)$ is a typing environment obtained by applying S to the type schemes of the bindings of Γ:

$$S(\Gamma) = \{(\mathtt{x} : S(\sigma)) \mid (\mathtt{x} : \sigma) \in \Gamma\}$$

We say Γ is more general than Γ' and write $\Gamma' \leq \Gamma$, if for the all bindings of Γ', Γ provides more general type schemes than Γ':

$$\Gamma' \leq \Gamma \iff \forall(\mathtt{x} : \sigma) \in \Gamma',\ \sigma \leq \Gamma(\mathtt{x})$$

2.5.2 Generalization

In our type system, we use type generalization with the value polymorphism restriction [Wri95]. This is not just because this restriction is used widely in real implementations of ML to solve the typing problems with reference cells, but also because it plays important role in the evaluation of extensional polymorphism. Consider the following program using a generic polymorphic primitive `input` whose type scheme is $\forall\delta.\mathtt{unit} \rightarrow \delta$.

```
let x = input () in x + x;;
```

This program should be regarded that it reads an integer and doubles the input integer. Without the value polymorphism restriction, however, the following type-annotated expression is regarded as well-typed:

```
let x{δ} = (input:unit → δ) () in (x:int) + (x:int);;
```

Because `x` would be defined as a derived polymorphic value whose type scheme is $\forall\delta.\delta$, it would be evaluated and stored in the evaluation environment as a generic closure

$$gclos(\emptyset, \_, \{\delta\}, (\texttt{input} : \texttt{unit} \to \delta)\ \texttt{()})$$

by (e-g-let), and not read a data at the moment of the definition of `x`. The actual evaluation of $(\texttt{input:unit} \to \delta)\ \texttt{()}$ would be done in the execution of (e-gvar) rule twice against both uses of `x`:

$$\frac{\emptyset \models \{\delta{\Rightarrow}\texttt{int}\}(\ (\texttt{input:unit} \to \delta)\ \ \texttt{()}\ ) \Longrightarrow v}{\{\texttt{x} = gclos(\ldots,\ \ (\texttt{input:unit} \to \delta)\ \ \texttt{()}\ )\} \models \texttt{x} : \texttt{int} \Longrightarrow v}$$

This means that for each `(x:int)` would call `input` once, therefore the whole expression would read two integers: this is not what we want.

This undesirable evaluation can be avoided if we introduce the value only polymorphism restriction. Generalization of let bindings are restricted to "*non expansive*" expressions. If the definition is expansive, its type is not generalized. The following predicate $generalizable(e_t)$ defines non expansive expressions:

$$\begin{array}{rcl}
generalizable(c_\sigma) & = & true \\
generalizable(\texttt{g}_\sigma) & = & true \\
generalizable(\texttt{x}) & = & true \\
generalizable(\texttt{fun x} \to e) & = & true \\
generalizable(e_1, e_2) & = & generalizable(e_1) \\
 & & \ \&\ generalizable(e_2) \\
generalizable(\texttt{let (rec) x} = e_1\ \texttt{in}\ e_2) & = & generalizable(e_1) \\
 & & \ \&\ generalizable(e_2) \\
\text{otherwise } generalizable(e) & = & false
\end{array}$$

Using *generalizable*, a generalization function *Gen* is defined. If e is generalizable (non-expansive), we permit the generalization of its type: we abstract the type variables which do not appear in the typing environment Γ freely.

$$\begin{array}{rl}
Gen(\Gamma, e, \tau) = & \text{if } generalizable(e) \\
 & \text{then } \forall\nu_1 \ldots \nu_n.\tau \text{ where } \{\nu_1, \ldots \nu_n\} = FV(\tau) \setminus FV(\Gamma) \\
 & \text{else } \forall\emptyset.\tau
\end{array}$$

We can extend this *Gen* for fully type-annotated expressions. The generalized type τ is the out-most type annotation of e_t:

$$Gen(\Gamma, e_t : \tau) = Gen(\Gamma, strip(e_t), \tau)$$

With the value polymorphism restriction, now the previous example has a typing like:

```
let x∅ = (input:unit → int) () in (x:int) + (x:int);;
```

and the evaluation of let definition is immediate using (e-let) rule. The evaluation of the definition expression will be the following:

$$\frac{\emptyset \models \texttt{()} \Longrightarrow \texttt{()} \quad \emptyset \models \texttt{input} : \texttt{unit} \rightarrow \texttt{int} \Longrightarrow clos(\emptyset, \_, \texttt{x}, e_t) \quad \{\texttt{x} = \texttt{()}\} \models e_t \Longrightarrow r}{\emptyset \models \texttt{(input : unit} \rightarrow \texttt{int) ()} \Longrightarrow r}$$

By (e-gen), the instance of `input` is evaluated to some function which takes `()` and returns an integer. In the evaluation of the whole expression, integer reading is performed just once and it returns the double of the input integer.

Before stepping to the typing rules, we prove a small lemma of the type generalization:

Lemma 5 (Type generalization and substitution) *Let Γ be a typing environment, e be a bare GML expression, τ be a type expression. For any substitution S,*

$$Gen(S(\Gamma), e : S(\tau)) \leq S(Gen(\Gamma, e : \tau))$$

PROOF: Let $\sigma = Gen(\Gamma, e : \tau)$ and $\sigma' = Gen(S(\Gamma), e : S(\tau))$.

If e is not generalizable, $S(\sigma) = \sigma' = S(\tau)$. Therefore $\sigma' \leq S(\sigma)$.

Otherwise, let $\{\nu_1, \ldots, \nu_n\} = FV(\tau) \setminus FV(\Gamma)$ then $\sigma = \forall \nu_1 \ldots \nu_n.\tau$. Let R be a substitution $\{\nu_i \Rightarrow \nu_i'\}$, which renames ν_i by fresh type variables ν_i'. Then, $S(\sigma) = \forall \nu_1' \ldots \nu_n'.S \circ R(\tau)$. $\sigma' = \forall V.S(\tau)$ where

$$V = FV(S(\tau)) \setminus FV(S(\Gamma)) = \bigcup_{\nu \in FV(\tau)} FV(S(\nu)) \setminus \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$$

Let τ' be a type expression such that $\tau' \leq_{S'} \sigma'$. Then $Dom(S') \subset V$ and $S' \circ S(\tau) = \tau'$.

Let $S'' = \{\nu_i' \Rightarrow S' \circ S(\nu_i)\}$. Then, for any ν_i,

$$S'' \circ S \circ R(\nu_i) = S'' \circ S(\nu_i') = S''(\nu_i') = S' \circ S(\nu_i)$$

since ν_i' are out of S.

For any $\nu \in FV(\tau) \setminus \{\nu_1, \ldots, \nu_n\}$, $\nu \in FV(\Gamma)$ and

$$S'' \circ S \circ R(\nu) = S'' \circ S(\nu) = S(\nu)$$

since $Dom(S'')$ is the set of fresh variables $\{\nu_1', \ldots, \nu_n'\}$. $S(\nu) = S' \circ S(\nu)$ since $Dom(S') \subset V$ and V is distinct from $\bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$.

From these, $S''(S \circ R(\tau)) = S' \circ S(\tau)$, which means $\tau' \leq_{S''} S(\sigma)$. □

2.5.3 Typing rules

We use *typing judgments* $\Gamma \vdash e_t$, which is read an expression e_t is well-typed under a type environment Γ. Because our expressions have already type annotations, we do not need to write the type of e_t like what we did at the definition of *Gen*. But sometimes we write it like $\Gamma \vdash e_t : \tau$ to make clear that the out-most type annotation of e_t is τ.

The typing rules are shown in Figure 2.1. Even though we have rather complicated semantics system with type annotation instantiations than those of the typical ML languages, our typing rules are as simple as the original and there are just few things to note.

$$\textbf{(t-cnst)}\ \frac{\tau \leq \sigma}{\Gamma \vdash c_\sigma : \tau}$$

$$\textbf{(t-gen)}\ \frac{\tau \leq \sigma}{\Gamma \vdash \mathtt{g}_\sigma : \tau}$$

$$\textbf{(t-inst)}\ \frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau}$$

$$\textbf{(t-fun)}\ \frac{\Gamma\{\mathtt{x} : \tau_1\} \vdash e_t : \tau_2}{\Gamma \vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2}$$

$$\textbf{(t-app)}\ \frac{\Gamma \vdash e_{t1} : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_{t2} : \tau_1}{\Gamma \vdash e_{t1}\ e_{t2} : \tau_2}$$

$$\textbf{(t-tpl)}\ \frac{\Gamma \vdash e_{t1} : \tau_1 \quad \Gamma \vdash e_{t2} : \tau_2}{\Gamma \vdash (e_{t1}, e_{t2}) : \tau_1 * \tau_2}$$

$$\textbf{(t-let)}\ \frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \qquad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

$$\textbf{(t-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1 \qquad \forall V.\tau_1 = Gen(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1) \\ \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let\ rec\ f}_V = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2}$$

Figure 2.1: Typing rules

Generic primitives $\mathtt{g}_\sigma$ is treated same as constants whose type schemes are σ in (t-gen).

The rule for recursive let definitions, (t-letrec), rejects non-functional definitions so that it can be isomorphic to their evaluation rule (e-letrec) which evaluates functions to recursive closures. Like the ordinary ML typing rules, the recursive identifier `f` is treated as monomorphic inside the typing of the definition $\mathtt{fun\ x} \rightarrow e_{t1}$. However, in the conclusion expression, the recursive definition $\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1$ is *not* as it is. We need to replace the type scheme annotations of the variable occurrence of `f` from $\mathtt{f}_{\tau_1}$ to $\mathtt{f}_{\forall V.\tau_1}$. This is for proper evaluation of recursive derived generic instances; recursive occurrences of `f` inside its definition must have dynamic polymorphic type scheme annotations, otherwise they cannot be evaluated by proper semantic rules for generic value instances (e-gvar*). Even though typing of the recursive occurrences of `f` is monomorphic, they must have polymorphic type annotations. The following *variable type annotation modifier* used in (t-letrec):

Let $Z = \{\mathtt{f}_\sigma \Rightarrow \mathtt{f}_{\sigma'}\}$, then

$$\begin{array}{rcl}
Z(\mathtt{f}_\sigma : \tau) & = & \mathtt{f}_{\sigma'} : \tau \\
Z(\mathtt{fun\ f} \rightarrow e_t : \tau) & = & \mathtt{fun\ f} \rightarrow e_t : \tau \\
Z(\mathtt{fun\ x} \rightarrow e_t : \tau) & = & \mathtt{fun\ x} \rightarrow Z(e_t) : \tau \\
Z(e_{t1}\ e_{t2}) & = & Z(e_{t1})\ Z(e_{t2}) \\
Z(e_{t1}, e_{t2}) & = & Z(e_{t1}), Z(e_{t2}) \\
Z(\mathtt{let\ (rec)\ f}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & \mathtt{let\ (rec)\ f}_V = e_{t1}\ \mathtt{in}\ Z(e_{t2}) : \tau \\
Z(\mathtt{let\ (rec)\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & \mathtt{let\ (rec)\ x}_V = Z(e_{t1})\ \mathtt{in}\ Z(e_{t2}) : \tau \\
\text{Otherwise, } Z(e_t) & = & e_t
\end{array}$$

Variable abstractions (functions and let bindings) guard the internal abstracted variables from the modifications.

Our typing judgments for fully type-annotated GML expressions can be transformed into typing judgments for bare GML expressions, in the familiar style of ML typing theory. If we have a bare GML expression e and if a type-annotated expression e_t such that $strip(e_t) = e$ has a typing $\Gamma \vdash e_t : \tau$, we write $\Gamma \vdash e : \tau$.

Example A bare GML expression

```
let double = fun x → plus x x in double;;
```

has the following typing, if we assume that `plus` is a generic primitive whose type scheme is $\forall\delta.\delta \rightarrow \delta \rightarrow \delta$ (here also some annotations are omitted):

$$\frac{\begin{array}{c}\emptyset \vdash \mathtt{fun\ x} \rightarrow \mathtt{(plus} : \delta \rightarrow \delta \rightarrow \delta\mathtt{)\ (x}{:}\delta\mathtt{)\ (x}{:}\delta\mathtt{)} : \delta \rightarrow \delta \\ \forall\delta.\delta \rightarrow \delta = Gen(\emptyset, \mathtt{fun\ x} \ldots : \delta \rightarrow \delta) \\ \{\mathtt{double} : \forall\delta.\delta \rightarrow \delta\} \vdash \mathtt{double} : \mathtt{int} \rightarrow \mathtt{int}\end{array}}{\begin{array}{ll}\emptyset \vdash & \mathtt{let\ double}_{\{\delta\}} = \mathtt{fun\ x} \rightarrow \\ & \quad \mathtt{(plus} : \delta \rightarrow \delta \rightarrow \delta\mathtt{)\ (x}{:}\delta\mathtt{)\ (x}{:}\delta\mathtt{)} : \delta \rightarrow \delta \\ & \mathtt{in\ double} : \mathtt{int} \rightarrow \mathtt{int}\end{array}}$$

The occurrences of δ inside the let definition expression are properly bound by the generalized type variable set attached at the declaration of `double`. □

Lemma 6 *Let e be a bare GML expression. Let Γ and Γ' be typing environments such that $\Gamma' \leq \Gamma$. If there is a typing $\Gamma' \vdash e : \tau$, $\Gamma \vdash e : \tau$.*

Proof: By induction on the structure of the typing delivations.

(t-inst) The expression must be a variable. Let $e = \mathtt{x}$. Let the typing as follows, for some type expression τ and type scheme σ':

$$\frac{\Gamma'(\mathtt{x}) = \sigma' \quad \tau \leq \sigma'}{\Gamma' \vdash \mathtt{x}_{\sigma'} : \tau}$$

From $\Gamma' \leq \Gamma$, we have $\sigma' \leq \Gamma(\mathtt{x})$. Let $\sigma = \Gamma(\mathtt{x})$. Since $\sigma' \leq \sigma$, $\tau \leq \sigma$. We can create a typing of $\mathtt{x}$ for Γ with the same type of the expression:

$$\frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq \sigma}{\Gamma \vdash \mathtt{x}_{\sigma} : \tau}$$

(t-const),(t-gen) Trivial since they make no use of typing environments.

The others They are proven easily using the induction.

□

Lemma 7 (Typing stability in recursive let bindings) *If an expression* $\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1$ *has a typing* $\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1$*, and* $\forall V.\tau_1 = \mathit{Gen}(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)$ *then*

$$\Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)$$

and $\forall V.\tau_1 = \mathit{Gen}(\Gamma, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1))$

PROOF: Each target variable occurrence of the annotation modification $\{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}$ must be typed under some $\Gamma'\{\mathtt{f} : \tau_1\}$, some typing environment of $\Gamma\{\mathtt{f} : \tau_1\}$ which have additional bindings to $\Gamma\{\mathtt{f} : \tau_1\}$, as

$$\Gamma'\{\mathtt{f} : \tau_1\} \vdash \mathtt{f} : \tau_1$$

by (t-inst). Because the type scheme of $\mathtt{f}$, τ_1 is completely monomorphic, the type of the variable occurrence must be also τ_1. Because $\tau_1 \leq \forall V.\tau_1$, The typing

$$\Gamma'\{\mathtt{f} : \forall V.\tau_1\} \vdash \mathtt{f} : \tau_1$$

is also correct from Lemma 6. This modification of the type scheme of $\mathtt{f}$ only affects the $\mathtt{f}$'s typings, and the other parts of typing are never modified. Therefore $\Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)$.

The generalization function *Gen* just looks the out-most type and the form of the whole expression, ignoring the others. Therefore replacing the type annotations of $\mathtt{f}$ does not affect the result of generalization. □

Our type system is stable under substitution:

Theorem 3 (Substitution lemma of the typing rules) *Typing is stable under substitution: if there is a typing $\Gamma \vdash e_t : \tau$, then for any admissible substitution S, $S(\Gamma) \vdash S(e_t) : S(\tau)$ also holds.*

PROOF: By induction over the structure of e_t.

Constants It is the case $e_t = c_\sigma$. The only applicative rule is (t-cnst), therefore we have $\Gamma \vdash c_\sigma : \tau$, where $\tau \leq_{S_0} \sigma$ for some substitution S_0. From Lemma 4,

$$S(\tau) \leq_{(S \circ S_0)|_{BV(\sigma)}} S(\sigma)$$

From the closedness of σ, we have $S(\sigma) = \sigma$. Using (t-cnst) we have $S(\Gamma) \vdash c_\sigma : S(\tau)$.

Generic primitives The proof is like one for constants, but using (t-gen) rule.

Variables Now $e_t = \mathtt{x}$, $\mathtt{x}$ has a typing $\Gamma \vdash \mathtt{x}_\sigma : \tau$ where $\tau \leq \sigma = \Gamma(\mathtt{x})$, and by Lemma 4, $S(\tau) \leq S(\sigma)$. Because $S(\Gamma)(\mathtt{x}) = S(\sigma)$, by using (t-inst) we can create the judgment $S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)$.

Let bindings The applicative rule is (t-let) and the judgment ends like:

$$\frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \quad \forall \nu_1 \ldots \nu_n.\tau_1 = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall \nu_1 \ldots \nu_n.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}_{\{\nu_1,\ldots,\nu_n\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

<u>Generalizable case</u>

If e_{t1} is generalizable, $\{\nu_1, \ldots, \nu_n\} = FV(\tau_1) \setminus FV(\Gamma)$ from the definition of *Gen*.

Let S' be a substitution which renames ν_i to ν'_i, new type variables which do not appear in any binding of S, Γ, nor the type annotations of e_{t1} and e_{t2}:

$$S' = \{\nu_i \Rightarrow \nu'_i\} \quad \nu'_i \text{ is a fresh variable}$$

Applying the induction hypothesis with this $S \circ S'$ to the first premise, we have:

$$(S \circ S')(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)$$

Since ν_i never appear in Γ freely, $S'(\Gamma) = \Gamma$. Therefore $(S \circ S')(\Gamma) = S(\Gamma)$ and

$$S(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)$$

Generalization of the type $(S \circ S')(\tau_1)$ is

$$\mathit{Gen}(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) = \forall V.(S \circ S')(\tau_1)$$

where $V = FV((S \circ S')(\tau_1)) \setminus FV(S(\Gamma))$.

We prove $V = \{\nu'_i\}$. From Lemma 1 ,

$$FV((S \circ S')(\tau_1)) = \bigcup_{\nu \in FV(\tau_1)} FV((S \circ S')(\nu))$$

$$FV(S(\Gamma)) = \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$$

and $\nu'_i \in FV((S \circ S')(\tau_1))$, because $\nu_i \in FV(\tau_1)$ and $(S \circ S')(\nu_i) = S(\nu'_i) = \nu'_i$. On the other hand $\nu'_i \notin FV(S(\Gamma))$, because ν'_i is a fresh variable. This proves $\{\nu'_i\} \subset V$.

Now we are going to prove the converse, $V \subset \{\nu'_i\}$. Suppose ν' is a variable in V, but not one of ν'_i. There must be a variable $\nu'' \in FV(\tau_1)$ such that $\nu' \in FV((S \circ S')(\nu''))$. ν'' cannot be one of ν_i, otherwise

$$(S \circ S')(\nu'') = (S \circ S')(\nu_i) = S(\nu'_i) = \nu'_i$$

and it would contradict our assumption that ν' is not one of ν'_i.

Since ν'' is not one of ν_i, $(S \circ S')(\nu'') = S(\nu'')$, hence $\nu' \in FV(S(\nu''))$. Since $\{\nu_i\} = FV(\tau_1) \setminus FV(\Gamma)$ and $\nu'' \in FV(\tau_1)$, we have $\nu'' \in FV(\Gamma)$. These lead to $\nu' \in \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu)) \subset FV(S(\Gamma))$. It contradicts our assumption that $\nu' \in V$.

Finally,

$$Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) = \forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1)$$

Because ν'_i's are fresh, they are out of the reach of S. Therefore $\forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1) = S(\forall \nu'_1 \ldots \nu'_n.S'(\tau_1))$. Since $\forall \nu'_1 \ldots \nu'_n.S'(\tau_1) \stackrel{\alpha}{=} \forall \nu_1 \ldots \nu_n.\tau_1$, the generalization is rewritten as follows:

$$Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) \stackrel{\alpha}{=} S(\forall \nu_1 \ldots \nu_n.\tau_1)$$

Now we use the induction hypothesis with S for the second premise and obtain:

$$S(\Gamma\{\mathtt{x} : \forall \nu_1 \ldots \nu_n.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2)$$

From these, we prove the following:

$$\frac{\begin{array}{c} S(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1) \\ \begin{array}{rcl} Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) & = & \forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1) \\ & \stackrel{\alpha}{=} & S(\forall \nu_1 \ldots \nu_n.\tau_1) \end{array} \\ S(\Gamma)\{\mathtt{x} : S(\forall \nu_1 \ldots \nu_n.\tau_1)\}) \vdash S(e_{t2}) : S(\tau_2) \end{array}}{S(\Gamma) \vdash \mathtt{let}\ \mathtt{x}_{\{\nu'_i\}} \mathtt{=} (S \circ S')(e_{t1})\ \mathtt{in}\ S(e_{t2}) : S(\tau_2)}$$

The conclusion expression is α-equivalent to $S(\mathtt{let}\ \mathtt{x}_{\{\nu_i\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2)$.

<u>Non generalizable case</u>

If $\mathit{generalizable}(e_{t1})$ is false, from the definition of Gen,

$$\sigma = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) = \tau_1$$

$$\mathit{Gen}(S(\Gamma), S(e_{t1}) : S(\tau_1)) = S(\tau_1)$$

therefore $S(\sigma) = S(\tau_1) = \mathit{Gen}(S(\Gamma), S(e_{t1}) : S(\tau_1))$ holds. From this we can prove the goal.

Recursive let bindings The proof goes same as the case of let bindings. The typing is obtained by (t-letrec):

$$\frac{\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1 \quad \forall V.\tau_1 = \mathit{Gen}(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1) \qquad \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2}{\Gamma \vdash \mathtt{let\ rec\ f}_V = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2}$$

We prepare a substitution S' which renames all the variables in V to fresh variables V', and use the induction hypothesis with $S \circ S'$ for the first premise:

$$(S \circ S')(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)$$

Because the variables in V never appear in Γ freely, $S'(\Gamma) = \Gamma$. Therefore we obtain:

$$S(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)$$

The generalization of the type $(S \circ S')(\tau_1)$ is

$$\mathit{Gen}(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = \forall V''.(S \circ S')(\tau_1)$$

where $V'' = FV((S \circ S')(\tau_1)) \setminus FV(S(\Gamma))$. We can prove $V' = V''$, as we did in the case of the non recursive let bindings. The generalization is rewritten as:

$$\mathit{Gen}(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = S(\forall V'.S'(\tau_1)) \overset{\alpha}{=} S(\forall V.\tau_1)$$

Now we use the induction hypothesis with S for the second premise:

$$S(\Gamma\{\mathtt{f} : \forall V.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2)$$

From these, using (t-letrec),

$$\frac{\begin{array}{c} S(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1) \\ \mathit{Gen}(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = \forall V'.(S \circ S')(\tau_1) \overset{\alpha}{=} S(\forall V.\tau_1) \\ S(\Gamma\{\mathtt{f} : \forall V.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2) \end{array}}{\begin{array}{l} S(\Gamma) \vdash \mathtt{let\ rec\ f}_{V'} = \\ \qquad \{\mathtt{f}_{S(\tau_1)} \Rightarrow \mathtt{f}_{\forall V'.(S \circ S')(\tau_1)}\}((S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}))\ \mathtt{in}\ S(e_{t2}) : S(\tau_2) \end{array}}$$

The conclusion expression is equal to

$$S(\texttt{let rec f}_{V'} = \{\texttt{f}_{\tau_1} \Rightarrow \texttt{f}_{\forall V'.S'(\tau_1)}\}(S'(\texttt{fun x} \to e_{t1}))\ \texttt{in}\ e_{t2} : \tau_2)$$

which is α-equivalent to

$$S(\texttt{let rec f}_{V} = \{\texttt{f}_{\tau_1} \Rightarrow \texttt{f}_{\forall V.\tau_1}\}(\texttt{fun x} \to e_{t1})\ \texttt{in}\ e_{t2} : \tau_2)$$

λ bindings, Applications, tuples The proofs of these cases are done in the same following manner: we have only one applicative rule for each construct, (t-fun), (t-app), and (t-tuple) respectively. We apply the induction hypothesis for the premises of each rule, then from these by using the same rule we can create the judgment $S(\Gamma) \vdash S(e_t) : S(\tau)$ easily.

□

2.5.4 Type soundness

A fully type-annotated expression e_t is called *well-typed* under a typing environment Γ, if it has a typing under Γ, i.e. $\Gamma \vdash e_t$. A *program* is an annotated expression e_t and it is evaluated under the empty evaluation environment. A program e_t is well-typed when e_t is a well-typed expression under the empty typing environment, $\emptyset \vdash e_t$. We reject the evaluation of the programs which lack typings. Contrary, if programs are well-typed, they are never evaluated to the error err:

Theorem 4 (Type soundness) *If a fully type-annotated expression e_t has a typing in the empty typing environment, $\emptyset \vdash e_t$, and the evaluation of e_t under the empty evaluation environment $\emptyset \models e_t$ terminates in some result r, then r is not err but a value.*

Theorem 4 does not assure termination of well-typed programs. It says that if a well-typed program terminates, the result must not be an error. Because of the existence of the recursive let definitions, evaluations can loop infinitely due to the (e-letrec) evaluation rule.

Since this property is too weak to prove directly, we need to generalize this theorem to something which also holds for the well-typed expressions under non-empty environments. For each evaluation derivation, we will prove that well-typed expression $e_t : \tau$ is not evaluated to err, but to some value v. Moreover, the return value v must be a semantic value belongs to the type τ. Hence for example we can make sure that the functional expressions are evaluated to functional semantic values (i.e. closures) in each step of the evaluation. For this purpose, we need to construct a typing system for the semantic values.

2.5.5 Typing of semantic values

In our *semantic typing system*, we have two kinds of judgments, $sv \in \sigma$ and $E \in \Gamma$. They mean that a semantic value sv has a semantic typing as σ, or an evaluation environment E has a semantic typing Γ. The semantic typing rules are in Figure 2.2.

The semantic typing rules for basic constants (s-bcnst), tuples (s-tpl), and the closures (s-clos*) and (s-gclos*) are coherent to their corresponding typing rules. In (s-clos-r), recursive functional closures may have polymorphic type schemes, since only the recursive let expressions can create recursive closures and its typing rule (t-letrec) reannotates defined identifiers with polymorphic type schemes. (s-cnst) and (s-gen) are our assumptions that the predefined semantic bindings of the constants and generic primitives are all semantically well-typed. This assumption for (s-gen) is actually too strong and may cause *coherence problem*. We will see it further in Section 2.5.6.

$$\textbf{(s-bcnst)}\ \frac{\tau \le \sigma}{b_\sigma \in \tau}$$

$$\textbf{(s-tpl)}\ \frac{v_1 \in \tau_1 \quad v_2 \in \tau_2}{v_1, v_2 \in \tau_1 * \tau_2}$$

$$\textbf{(s-clos)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma \vdash \texttt{fun x} \to e_t : \tau_1 \to \tau_2}{clos(E, \_, \texttt{x}, e_t) \in \tau_1 \to \tau_2}$$

$$\textbf{(s-clos-r)}\ \frac{\begin{array}{c}\exists\Gamma, \quad E \in \Gamma \quad \Gamma\{\texttt{f} : \forall V.\tau_1 \to \tau_2\} \vdash \texttt{fun x} \to e_t : \tau_1 \to \tau_2 \\ \forall V.\tau_1 \to \tau_2 = Gen(\Gamma, \texttt{fun x} \to e_t) \quad V \cap \mathbb{D} = \emptyset\end{array}}{clos(E, \texttt{f}, \texttt{x}, e_t) \in \forall V.\tau_1 \to \tau_2}$$

$$\textbf{(s-gclos)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \ne \emptyset}{gclos(E, \_, V, e_t : \tau) \in \forall V.\tau}$$

$$\textbf{(s-gclos-r)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma\{\texttt{x} : \forall V.\tau\} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \ne \emptyset}{gclos(E, \texttt{x}, V, e_t : \tau) \in \forall V.\tau}$$

$$\textbf{(s-env)}\ \frac{\forall(\texttt{x} : \sigma) \in \Gamma, \quad E(\texttt{x}) = sv \quad sv \in \sigma}{E \in \Gamma}$$

$$\textbf{(s-schm)}\ \frac{\forall\tau \le_S \sigma, \quad S(v) \in \tau}{v \in \sigma}$$

$$\textbf{(s-cnst)}\ \frac{\tau \le_S \sigma}{S(cnst(c_\sigma)) \in \tau}$$

$$\textbf{(s-gen)}\ \frac{\tau \le \sigma}{gen(\mathsf{g}_\sigma, \tau) \in \tau}$$

Figure 2.2: The semantic typing rules

Lemma 8 (Substitution lemma of semantic typing) *If a semantic value sv has a semantic typing $sv \in \sigma$, then $S(sv) \in S(\sigma)$ for any substitution S. If we have an evaluation environment E such that $E \in \Gamma$ for some typing environment Γ, for any substitution S, $S(E) \in S(\Gamma)$.*

PROOF: By induction on the typing derivation of sv.

(s-bcnst) We have $b_\sigma \in \tau$ and $\tau \le \sigma$. Because σ is closed, for any substitution S, $S(\tau) \le S(\sigma) = \sigma$, therefore using (s-bcnst) and we obtain $S(b_\sigma) = b_\sigma \in S(\tau)$.

(s-tpl) Let $v = (v_1, v_2)$. Using the induction hypothesis we have $S(v_i) \in S(\tau_i)$. Applying (s-tpl) against these, we get $S(v_1, v_2) \in S(\tau_1 * \tau_2)$.

(s-clos) Suppose that

$$v = clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \rightarrow \tau_2$$

This requires that we have some typing environment Γ such that $E \in \Gamma$ and $\Gamma \vdash \mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2$. By Theorem 3,

$$S(\Gamma) \vdash S(\mathtt{fun\ x} \rightarrow e_t) : S(\tau_1 \rightarrow \tau_2)$$

Using the induction hypothesis for $E \in \Gamma$, we have $S(E) \in S(\Gamma)$. From these we use (s-clos) and obtain:

$$clos(S(E), \_, \mathtt{x}, S(e_t)) = S(clos(E, \_, \mathtt{x}, e_t)) \in S(\tau_1 \rightarrow \tau_2)$$

(s-clos-r) Suppose

$$v = clos(E, \mathtt{f}, \mathtt{x}, e_t) \in \forall V.\tau_1 \rightarrow \tau_2$$

then from this semantic typing, we have Γ such that $E \in \Gamma$ and $\Gamma\{\mathtt{f} : \forall V.\tau_1 \rightarrow \tau_2\} \vdash \mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2$, where $\forall V.\tau_1 \rightarrow \tau_2 = Gen(\Gamma, \mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2)$. We rename the variables in V appear in the typing to the fresh ones V' by using a substitution $S' = \{\nu_i \Rightarrow \nu'_i \mid \nu_i \in V\}$. We also α-convert the type scheme of $\mathtt{f}$. In the same way as we did in the substitution lemma of let typing, we have:

$$S'(\Gamma)\{\mathtt{f} : \forall V'.S'(\tau_1 \rightarrow \tau_2)\} \vdash S'(\mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2)$$
$$\forall V'.S'(\tau_1 \rightarrow \tau_2) = Gen(S'(\Gamma), S'(\mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2))$$

Because V is a set of variables which never appear in Γ freely, $S'(\Gamma) = \Gamma$.

Using Theorem 3 with the fact that V' are fresh and out of the reach of S,

$$S(\Gamma)\{\mathtt{f} : \forall V'.(S \circ S')(\tau_1 \rightarrow \tau_2)\} \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2)$$
$$\forall V'.(S \circ S')(\tau_1 \rightarrow \tau_2) = Gen(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow (e_t : \tau_2) : \tau_1 \rightarrow \tau_2))$$

From the induction hypothesis for $E \in \Gamma$, we have $S(E) \in S(\Gamma)$. We use (s-clos-r) and obtain:

$$clos(S(E), \mathtt{f}, \mathtt{x}, S(e_t)) = S(clos(E, \mathtt{f}, \mathtt{x}, e_t)) \in \forall V'.(S \circ S')(\tau_1 \rightarrow \tau_2)$$

where $\forall V'.(S \circ S')(\tau_1 \rightarrow \tau_2) \stackrel{\alpha}{=} S(\forall V.\tau_1 \rightarrow \tau_2)$.

(s-gclos)

$$\frac{\exists\Gamma,\quad E \in \Gamma \quad \Gamma \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau)}{gclos(E, \_, V, e_t : \tau) \in \forall V.\tau}$$

Like the part of the let bindings of the proof of Theorem 3, we prepare a substitution S' which renames all the type variables in V to fresh variables:

$$S' = \{(\nu_i \Rightarrow \nu_i') \mid \nu_i \in V\} \text{ where } \nu_i'\text{'s are fresh}$$

and let $V' = \{\nu_i'\}$. By Theorem 3, we have $(S \circ S')(\Gamma) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$ and $(S \circ S')(\Gamma) = S(\Gamma)$. Therefore:

$$S(\Gamma) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$$

The generalization of this $(S \circ S')(\tau)$ is $Gen(S(\Gamma), (S \circ S')(e_t) : (S \circ S')(\tau))$ and it is equal to $\forall V'.(S \circ S')(\tau) \stackrel{\alpha}{=} S(\forall V.\tau)$ also by the same discussion of Theorem 3. Using the induction hypothesis with S for $E \in \Gamma$ and (s-gclos), we have

$$\frac{\begin{array}{c} S(E) \in S(\Gamma) \quad S(\Gamma) \vdash (S \circ S')(e_t) \\ S(\forall V.\tau) \stackrel{\alpha}{=} \forall V'.(S \circ S')(\tau) = Gen(S(\Gamma), (S \circ S')(e_t)) \end{array}}{gclos(S(E), \_, V', (S \circ S')(e_t)) \in S(\forall V.\tau)}$$

The generic closure in the conclusion is α-equivalent to $S(gclos(E, \_, V, e_t))$.

(s-gclos-r) This is almost same as (s-gclos).

$$\frac{\exists\Gamma,\quad E \in \Gamma \quad \Gamma\{\mathtt{x} : \tau\} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau)}{gclos(E, \mathtt{x}, V, e_t : \tau) \in \forall V.\tau}$$

Similar to (s-gclos), let S' be a substitution which renames all the type variables in V to fresh variables: $S' = \{(\nu_i \Rightarrow \nu_i') \mid \nu_i \in V\}$ and let $V' = \{\nu_i'\}$. By Theorem 3, we have $(S \circ S')(\Gamma\{\mathtt{x} : \tau\}) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$ and $(S \circ S')(\Gamma) = S(\Gamma)$. Therefore:

$$S(\Gamma)\{\mathtt{x} : (S \circ S')(\tau)\} \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$$

The generalization $(S \circ S')(\tau)$ is $Gen(S(\Gamma), (S \circ S')(e_t) : (S \circ S')(\tau)) = \forall V'.(S \circ S')(\tau) \stackrel{\alpha}{=} S(\forall V.\tau)$. Using the induction hypothesis with S for $E \in \Gamma$ and (s-gclos), we have

$$\frac{\begin{array}{c} S(E) \in S(\Gamma) \quad S(\Gamma)\{\mathtt{x} : (S \circ S')(\tau)\} \vdash (S \circ S')(e_t) \\ S(\forall V.\tau) \stackrel{\alpha}{=} \forall V'.(S \circ S')(\tau) = Gen(S(\Gamma), (S \circ S')(e_t)) \end{array}}{gclos(S(E), \mathtt{x}, V', (S \circ S')(e_t)) \in S(\forall V.\tau)}$$

The generic closure in the conclusion is α-equivalent to $S(gclos(E, \mathtt{x}, V, e_t))$.

(s-env) For any $(\mathtt{x} : \sigma) \in \Gamma$, there is a semantic value sv such that $E(\mathtt{x}) = sv$ and $sv \in \sigma$. Using the induction hypothesis, we have $S(E(\mathtt{x})) = S(sv) \in S(\sigma)$ From it and $(\mathtt{x} : S(\sigma)) \in S(\Gamma)$,

$$S(E) \in S(\Gamma)$$

(s-schm) We have

$$\frac{\forall \tau \leq_{S_0} \sigma \quad S_0(v) \in \tau}{v \in \sigma}$$

and we can let $\sigma = \forall \nu_1 \ldots \nu_n.\tau'$ so that ν_i never appears in the bindings of S, without losing any generality. Any instance τ'' of $S(\sigma)$, $\tau'' \leq_{S''} S(\sigma) = \forall \nu_1 \ldots \nu_n.S(\tau')$, τ'' is written as $(S'' \circ S)(\tau')$, therefore $\tau'' \leq_{(S'' \circ S)} \sigma$. By induction hypothesis,

$$S''(S(v)) \in \tau''$$

Since this holds for any instance τ'' of $S(\sigma)$, $\tau'' \leq_{S''} S(\sigma)$, using (s-schm) and this result, we have

$$S(v) \in S(\sigma)$$

$\square$

Lemma 9 (Generality of semantic typing) *If a value v has a semantic typing $v \in \tau$, $v \in \forall V.\tau$ where $V \subset FV(\tau)$.*

Proof: It is clear from Lemma 8 and (s-schm) rule. $\square$

With the semantic typing system, we can enforce Theorem 4 and obtain the following theorem which can be proven by induction over the evaluation steps.

Theorem 5 (Stronger type soundness) *If $\Gamma \vdash e_t : \tau$ for some Γ and there exists E such that $E \in \Gamma$ and $E \models e_t : \tau \Longrightarrow r$, then r is not err but a value v and $v \in \tau$ holds.*

Proof: By induction over the size of evaluation derivations $E \models e_t \Longrightarrow r$.

Constants Let $e_t = c_\sigma : \tau$, then its typing is given by (t-cnst):

$$\frac{\tau \leq_S \sigma}{\Gamma \vdash c_\sigma : \tau}$$

Constants are evaluated by (e-cnst) rather explicitly using the function *cnst*:

$$\frac{\tau \leq_S \sigma}{E \models c_\sigma : \tau \Longrightarrow S(cnst(c_\sigma))}$$

As we have assumed in (s-cnst), the result $S(cnst(c_\sigma))$ must return a semantic value who belongs to the type τ.

Generic primitives Generic primitives are evaluated by (e-gen) rule:

$$E \models \mathsf{g}_\sigma : \tau \Longrightarrow gen(\mathsf{g}, \tau)$$

From the assumption (s-gen), $gen(\mathsf{g}, \tau) \in \tau$.

Tuples We have a typing

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \quad \Gamma \vdash e_{t2} : \tau_2}{\Gamma \vdash (e_{t1}, e_{t2}) : \tau_1 * \tau_2}$$

and the possible evaluation rules are (e-tpl) and (e-tpl-err), but (e-tpl-err) contradicts the induction hypothesis; one of the sub-expression e_{ti} would be evaluated to err whose evaluation size is smaller than the tuple itself.

Therefore (e-tpl) is the only applicable rule:

$$\frac{E \models e_{t1} \Longrightarrow v_1 \quad E \models e_{t2} \Longrightarrow v_2}{E \models (e_{t1}, e_{t2}) : \tau \Longrightarrow (v_1, v_2)}$$

Using the induction hypothesis for each tuple content e_{ti} we have $v_i \in \tau_i$, from these by (s-tuple),

$$(v_1, v_2) \in \tau_1 * \tau_2$$

Variables For variables we have several rules, but we do not need to think about (e-var-err), because from $\Gamma \vdash \mathtt{x}_\sigma : \tau$, $\tau \leq \sigma$ such that $\Gamma(\mathtt{x}) = \sigma$ and $E \in \Gamma$, $E(\mathtt{x}) = sv$ for some semantic value sv and $sv \in \sigma$.

The rule we can use depends on the form of sv. If sv is a value v, it must be (e-nvar) because $\tau \leq_S \sigma$ for some S, and from it we have

$$E \models \mathtt{x} : \tau \Longrightarrow v$$

Since $\tau \leq_S \sigma$ and $v \in \sigma$, from (s-schm) $v \in \tau$ holds.

If sv is a non-recursive generic closure, from (s-gclos),

$$sv = gclos(E', \_, V, e_t : \tau') \in \sigma = \forall V.\tau'$$

and the following things hold:

$$\exists \Gamma' \quad E' \in \Gamma' \quad \forall V.\tau' = Gen(\Gamma', e_t : \tau')$$

Because $\tau \leq \sigma$, (e-gvar-err) is never applied. Only (e-gvar) is available:

$$\frac{\exists S,\ \tau \leq_S \forall V.\tau' \quad E' \models S(e_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x} : \tau \Longrightarrow r}$$

Because $Dom(S) = V$ and none of the variables in V appear freely in Γ' from the generalization, $S(\Gamma') = \Gamma'$ and $S(E') = E'$. Therefore $E' = S(E') \in S(\Gamma') = \Gamma'$ and $S(\Gamma') \vdash S(e_t : \tau')$ from Theorem 3. We can use the induction hypothesis for $S(e_t)$:

$$E' \models S(e_t : \tau') \Longrightarrow v \in S(\tau')$$

Since $S(\tau') = \tau$, we have $v \in \tau$. And finally we can prove

$$E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x} : \tau \Longrightarrow v \in \tau$$

The case if v is a recursive generic closure, $gclos(E', \mathtt{f}, V, e_t)$, is also proven in the same way. Instead of (e-gvar), we use (e-gvar-r).

Functions The typing is

$$\frac{\Gamma\{\mathtt{x} : \tau_1\} \vdash e_t : \tau_2}{\Gamma \vdash \mathtt{fun}\ \mathtt{x} \to e_t : \tau_1 \to \tau_2}$$

and the last evaluation step must use (e-fun):

$$E \models \mathtt{fun}\ \mathtt{x} \to e_t \Longrightarrow clos(E, \_, \mathtt{x}, e_t)$$

where $E \in \Gamma$. From them we can apply (s-clos) and obtain

$$clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \to \tau_2$$

Let bindings We have the following typing

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

The possible evaluation rules are dependent on whether $V \cap \mathbb{D} = \emptyset$ or not.

<u>Normal let</u>

If $V \cap \mathbb{D} = \emptyset$, it is a normal let definition which defines a non-generic value. The possible rules are (e-let) and (e-let-err). From the same reasoning we have done for tuples, we can say that (e-let-err) cannot be used and (e-let) is the only applicable one.

$$\frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} : \tau_1 \Longrightarrow v_1 \quad E\{\mathtt{x} = v_1\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V = (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau \Longrightarrow r}$$

Using the induction hypothesis for the evaluation of this e_{t1}, we have $v_1 \in \tau_1$. From Lemma 9, $v_1 \in \forall V.\tau_1$ and $E\{\mathtt{x} = v_1\} \in \Gamma\{\mathtt{x} : \forall V.\tau_1\}$, therefore we can use the induction hypothesis again for the evaluation of e_{t2}:

$$E\{\mathtt{x} = v_1\} \models e_{t2} : \tau_2 \Longrightarrow v_2 \in \tau_2$$

And from $v_2 \in \tau_2$, we prove

$$E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau_2 \Longrightarrow v \in \tau_2$$

<u>Generic let</u>

If V contains dynamic variables, $V \cap \mathbb{D} \neq \emptyset$, the expression defines a derived generic value. We have (e-g-let) as the only applicable rule:

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \_, V, e_{t1} : \tau_1)\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau_2 \Longrightarrow r}$$

From (s-gclos), $gclos(E, \_, V, e_{t1} : \tau_1) \in \forall V.\tau_1$ and therefore

$$E\{\mathtt{x} = gclos(E, \_, V, e_{t1} : \tau_1)\} \in \Gamma\{\mathtt{x} : \forall V.\tau_1\}$$

This permits us to use the induction hypothesis for the evaluation of e_{t2}. It must return a value $v \in \tau_2$. And it is our goal.

Recursive let bindings We can prove in the same way as the non-recursive let bindings, using the recursive version of rules and the rule for the functions. The typing of the expression must be as follows by (t-letrec) with the expression e'_{t1} such that $e_{t1} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\} e'_{t1}$:

$$\frac{\begin{array}{c}\Gamma\{\mathtt{f} : \tau_1\} \vdash \mathtt{fun}\ \mathtt{x} \rightarrow e'_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma\{\mathtt{f} : \tau_1\}, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2 \quad e_{t1} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\} e'_{t1}\end{array}}{\Gamma \vdash \mathtt{let\ rec}\ \mathtt{f}_V \,\mathtt{=}\, \mathtt{fun}\ \mathtt{x} \rightarrow e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

<u>Normal let</u>

If $V \cap \mathbb{D} = \emptyset$, the definition is not a generic one and its evaluation is done by (e-letrec). (e-letrec-err) cannot be applied.

$$\frac{V \cap \mathbb{D} = \emptyset \quad E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec}\ \mathtt{f}_V \,\mathtt{=}\, \mathtt{fun}\ \mathtt{x} \rightarrow e_{t1} : \tau_1\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

By (s-clos-r), $clos(E, \mathtt{f}, \mathtt{x}, e_{t1}) \in \forall V.\tau_1$. Therefore

$$E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \in \Gamma\{\mathtt{f} : \forall V.\tau_1\}$$

and we can use the induction hypothesis again for the evaluation of e_{t2}:

$$E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} : \tau_2 \Longrightarrow v_2 \in \tau_2$$

this leads to the evaluation of the whole expression:

$$E \models \texttt{let rec f}_V\texttt{ = fun x} \to e_{t1} \texttt{ in } e_{t2} : \tau_2 \Longrightarrow v \in \tau_2$$

<u>Generic let</u>

If V contains dynamic variables i.e. $V \cap \mathbb{D} \neq \emptyset$, the expression defines a derived generic value. We have (e-g-letrec) as the only applicable evaluation rule:

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\texttt{x} = gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1)\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \texttt{let rec x}_V\texttt{ = fun x} \to e_{t1} : \tau_1 \texttt{ in } (e_{t2} : \tau_2) : \tau_2 \Longrightarrow r}$$

From (s-gclos-r), $gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1) \in \forall V.\tau_1$ and therefore

$$E\{\texttt{x} = gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1)\} \in \Gamma\{\texttt{x} : \forall V.\tau_1\}$$

This permits us to use the induction hypothesis to prove the safe evaluation of e_{t2} to $v \in \tau_2$.

Application The typing is

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \to \tau_2 \quad \Gamma \vdash e_{t2} : \tau_1}{\Gamma \vdash (e_{t1} : \tau_1 \to \tau_2)\ (e_{t2} : \tau_1) : \tau_2}$$

and the evaluation of the argument e_{t2} is proven successful in the same way we did for tuples. Therefore (e-app-err1) is never used. (e-app-err2) is neither possible, because using the induction hypothesis, $e_{t1} : \tau_1 \to \tau_2$ must return a closure from (s-clos).

If the last used evaluation rule is (e-app1), e_{t1} is evaluated to non-recursive closure:

$$\frac{\begin{array}{c} E \models e_{t2} : \tau_1 \Longrightarrow v_2 \\ E \models e_{t1} : \tau_1 \to \tau_2 \Longrightarrow clos(E', \_, \texttt{x}, e_{t3} : \tau_3) \\ E'\{\texttt{x} = v_2 : \tau_1\} \models e_{t3} : \tau_3 \Longrightarrow r \end{array}}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

for some $e_{t3} : \tau_3$. Using the induction hypothesis for the first two premises, we have

$$v_2 \in \tau_1$$

$$clos(E', \_, \texttt{x}, e_{t3} : \tau_3) \in \tau_1 \to \tau_2$$

From (s-clos), $\tau_3 = \tau_2$ and there must be a type environment Γ' such that $E' \in \Gamma'$ and

$$\Gamma' \vdash \texttt{fun x} \to e_{t3} : \tau_1 \to \tau_2$$

This typing must end with (t-fun):

$$\frac{\Gamma'\{\texttt{x} : \tau_1\} \vdash e_{t3} : \tau_2}{\Gamma' \vdash \texttt{fun x} \to e_{t3} : \tau_1 \to \tau_2}$$

From this and $v_2 \in \tau_1$, $E'\{\mathtt{x} = v_2\} \in \Gamma'\{\mathtt{x} : \tau_1\}$. We can now use the induction hypothesis for the last premise and obtain:

$$E'\{\mathtt{x} = v_2\} \models e_{t3} : \tau_2 \Longrightarrow v \in \tau_2$$

for some value v. This ensures the evaluation of the application itself:

$$E \models e_{t1}\ e_{t2} \Longrightarrow v \in \tau_2$$

If the last evaluation rule is (e-app2), e_{t1} is evaluated to a recursive closure, but the outline of the proof is as same as non-recursive version.

□

Theorem 4 is now just a special case of Theorem 5, when the both typing and evaluation environments are empty sets.

2.5.6 Type soundness of generic primitives

In Section 2.5.5, we have assumed that a generic primitive g_σ has a well-defined semantic value for *any* instance type τ of σ. But it is not actually true, since τ may contain free type variables. Let $\sigma = \forall\delta.\delta$ and $\tau = \delta'$ for instance. Then, $gen(\mathsf{g}, \delta')$ must return a semantic value of δ' which has no actual semantic value in practice. Without semantic values of δ', it is impossible to evaluate $\mathsf{g} : \delta'$ and the type soundness is broken.

To overcome this undesirable situation we can take two possible approaches: the first is to ignore the problem at static typing phase and leave such instances without semantics raise run-time exceptions. When the primitive function $gen(\mathsf{g}, \tau)$ finds that the type τ has no actual semantic value, it is responsibility of the primitive to raise a run-time exception which can be caught by an exception handler.

Another solution is to weaken the assumption of gen. We now assume that $gen(\mathsf{g}, \tau)$ returns a semantic value of τ only when τ is closed. The rule (s-gen) is rewritten as follows:

$$\textbf{(s-gen)}\ \frac{\tau \leq \sigma \quad FV(\tau) = \emptyset}{gen(\mathsf{g}_\sigma, \tau) \in \tau}$$

If any basic constant type contains at least one semantic value, we can construct a well-typed semantic value of any closed type τ. Therefore $gen(\mathsf{g}, \tau)$ can always return some semantic value of any closed type τ[2]. However, this weakened assumption requires a new restriction to generic primitives: for the type soundness, a generic primitive must not be evaluated under an evaluation type context with type variables. In type system, this is reflected to a type instantiation restriction of generic values: type annotation of generic instances must not contain free dynamic type variables. For example, the expression

[2] *gen* still needs to raise a run-time exception when it is asked to create values of abstract data types whose definitions are unknown to *gen*.

```
(g∀δ.δ : unit → δ') ();;
```

must be ill-typed, since the type of g contains a free dynamic type variable δ'. On the other hand, to define a derived generic value, the same expression must be well-typed when the dynamic type variable δ' is abstracted by some generalization as below:

```
let derived{δ'} () = (g∀δ.δ : unit → δ') () in ...
```

As generic primitives, derived generic values must be under this type restriction, too. Unfortunately, the core type system as a simple extension of Damas-Milner's is not powerful enough to distinguish free and generalized type variables. Therefore, in this chapter, we keep using the original assumption of *gen* that $gen(\mathtt{g}, \tau)$ always returns some semantic value of τ. We will handle this new typing restriction later when we build *generic constraint typing system* in Chapter 8, that detects illegal uses of generic instances, which is independent of the core typing. In the generic constraint typing system, a type judgment $\Gamma, V \Vdash e_t$ carries a set of *generalized type variables*. Type variables in V are considered to be generalized in following typing derivation, and the free occurrences of dynamic type variables in e_t are restricted within those of V.

The existence of free dynamic type variables in typing also causes *coherence problem* of generic primitives. This will be discussed in Section 3.6 in the chapter of the compilation.

2.6 Type reconstruction

Since the semantics of GML cannot be defined without type annotations, to evaluate GML programs, we require that they must be fully type-annotated. But writing programs with fully type annotations by hand as explicit type constraints is a boring job. We definitely need to have a type reconstruction which attaches type annotations to bare GML expressions which are correct in respect with the typing rules, like other relatives of the ML family.

We here represent a type reconstruction algorithm for GML so that we can write programs without type annotations. The algorithm is based on one of Milner-Hindley's for ML. It will verify the correctness of expressions and return their fully type-annotated versions, if they have some typings. The returned typings are always "principal", which are more general than the other possible typings. If they have no possible typing at all, the algorithm will detect it and report typing errors. By using our type reconstruction algorithm, we do not need to write explicit type constraints in almost cases.

2.6.1 Instantiation

Algorithm *inst* creates an instance of a type scheme, whose quantified type variables are all instantiated by different fresh variables. Distinction of regular and dynamic type variables are important here also. *inst* must return an admissible substitution: a regular quantified type variable must be instantiated to a regular variable and a dynamic variable must be instantiated to a dynamic variable.

$inst(\sigma) =$
  let $\forall \nu_1 \ldots \nu_n.\tau = \sigma$
  let ν'_i be a regular or dynamic fresh variable according to ν_i
  let $S = \{\nu_i \Rightarrow \nu'_i\}$
  in $S(\tau)$

Clearly $inst(\sigma)$ is one of the most general instances of σ:

Theorem 6 (Generality of *inst*) *Let $\tau' = inst(\sigma)$. For any instance τ'' of σ, there exists a substitution S such that $S(\tau') = \tau''$.*

PROOF: Let $\sigma = \forall \nu_1 \ldots \nu_n.\tau$. Then $inst(\sigma) = \{\nu_i \Rightarrow \nu'_i\}$ for some fresh variables ν'_i. Suppose $\tau'' \leq_S \sigma$. Then

$$(S \circ \{\nu'_i \Rightarrow \nu_i\})(\tau') = (S \circ \{\nu'_i \Rightarrow \nu_i\} \circ \{\nu_i \Rightarrow \nu'_i\})(\tau) = S(\tau) = \tau''$$

□

Lemma 10 (Substitution lemma of *inst*) *If $inst(\sigma) = \tau$, then for any substitution S where $BV(\sigma)$ is out of the reach of S, $inst(S(\sigma)) = S(\tau)$.*

PROOF: Let $\sigma = \forall \nu_1 \ldots \nu_n.\tau_0$ and $\{\nu_i \Rightarrow \nu'_i\} = inst(\sigma)$. Since $\{\nu_i\}$ is out of S, $S(\sigma) = \forall \nu_1 \ldots \nu_n.S(\tau_0)$. We can suppose that ν'_i are selected so that they are also out of the reach of S. $\{\nu_i \Rightarrow \nu'_i\} \circ S = \{\nu_i \Rightarrow \nu'_i\} + S = S \circ \{\nu_i \Rightarrow \nu'_i\}$. From it, $inst(S(\sigma)) \stackrel{\alpha}{=} \{\nu_i \Rightarrow \nu'_i\}(S(\tau_0)) = S(\{\nu_i \Rightarrow \nu'_i\}(\tau_0)) = S(\tau)$ □

2.6.2 Unification

If for two types τ_1 and τ_2, there exists a substitution U such that $U(\tau_1) = U(\tau_2)$, then we say the types are *unifiable* and call U a *unifier* of τ_1 and τ_2. Our type unification algorithm seeks the *most general unifier* of given two types, if it exists. The returned unifiers must be admissible: they map dynamic type variables to type expressions without regular type variables. If such U exists, we say τ_1 and τ_2 are *unifiable*. When we unify a type expression τ with a dynamic type variable δ, we must *dynamize* the expression in order to have admissible unifiers. The dynamization algorithm $dynamize(\tau)$ returns a substitution which replaces all the regular type variables α_i in τ to dynamic type variables δ_i keeping their distinction:

$$\begin{array}{rcl}
dynamize(base) & = & \emptyset \\
dynamize(\alpha) & = & \{\alpha \Rightarrow \delta\} \text{ where } \delta \text{ is a fresh dynamic type variable} \\
dynamize(\delta) & = & \emptyset \\
dynamize(\tau_1 \rightarrow \tau_2) & = & \text{let } D_1 = dynamize(\tau_1),\ D_2 = dynamize(D_1(\tau_1)) \text{ in } D_2 \circ D_1 \\
dynamize(\tau_1 * \tau_2) & = & \text{let } D_1 = dynamize(\tau_1),\ D_2 = dynamize(D_1(\tau_1)) \text{ in } D_2 \circ D_1
\end{array}$$

For example,

$$dynamize(\alpha * \delta_0 \rightarrow \beta) = \{\alpha \Rightarrow \delta_1, \beta \Rightarrow \delta_2\}$$

and applying this to the type $\alpha * \delta_0 \rightarrow \beta$, we get its dynamized type expression:

$$\{\alpha \Rightarrow \delta_1, \beta \Rightarrow \delta_2\} (\alpha * \delta_0 \rightarrow \beta) = \delta_1 * \delta_0 \rightarrow \delta_2$$

With *dynamize* we can define the algorithm *unify* to create the most general unifier of two types. This is a variant of Robinson's algorithm [Rob65] plus the dynamization for unification between a dynamic type variable and a type expression:

$$\begin{array}{rcl}
unify(\tau, \tau) & = & \emptyset \\
unify(\alpha, \tau) & = & \{\alpha \Rightarrow \tau\} \text{ when } \alpha \notin FV(\tau) \\
unify(\delta, \tau) & = & D \circ \{\delta \Rightarrow D(\tau)\} \text{ when } \delta \notin FV(\tau), \text{ where } D = dynamize(\tau) \\
unify(\tau, \nu) & = & unify(\nu, \tau)
\end{array}$$

$$\begin{array}{l}
unify(\tau_1 \rightarrow \tau_2, \tau_1' \rightarrow \tau_2') = \\
\qquad\qquad \text{let } U_1 = unify(\tau_1, \tau_1'),\ U_2 = unify(U_1(\tau_2), U_1(\tau_2')) \text{ in } U_2 \circ U_1 \\
unify(\tau_1 * \tau_2, \tau_1' * \tau_2') = \\
\qquad\qquad \text{let } U_1 = unify(\tau_1, \tau_1'),\ U_2 = unify(U_1(\tau_2), U_1(\tau_2')) \text{ in } U_2 \circ U_1 \\
unify(\tau_1, \tau_2) \text{ fails if } \tau_1 \text{ and } \tau_2 \text{ do not match with one of these above}
\end{array}$$

Theorem 7 *If $unify(\tau_1, \tau_2)$ succeeds, it returns an admissible substitution.*

PROOF: It is clear from the definition of *dynamize*, that for any type τ, the domain of $dynamize(\tau)$ contains no dynamic type variable. Therefore $dynamize(\tau)$ always returns an admissible substitution.

In the algorithm *unify*, the third rule $unify(\delta, \tau) = \ldots$ is the only case which introduce a substitution whose domain contains dynamic type variable. $D \circ \{\delta \Rightarrow D(\tau)\} = D \cup \{\delta \Rightarrow D(\tau)\}$, because $Dom(D)$ has only regular type variables while $D(\tau)$ contains no regular type variable. Because δ is the only dynamic type variable of $Dom(unify(\delta, \tau))$ and it maps δ to $D(\tau)$ which has no regular type variable, it returns an admissible substitution.

From Theorem 2, admissibility is kept in composition of substitutions, the algorithm *unify* always returns an admissible substitution, if it succeeds. □

Theorem 8 (Unification algorithm returns MGU) *The algorithm $unify(\tau_1, \tau_2)$ returns one of their most general unifier, if they are unifiable. If they are unifiable with a unifier S, i.e. $S(\tau_1) = S(\tau_2)$, the algorithm $unify(\tau_1, \tau_2)$ returns a substitution U and $S = S' \circ U$ for some S'.*

Otherwise, in the case they are not unifiable, $unify(\tau_1, \tau_2)$ fails.

PROOF: The proof is by induction over the structure of the argument type expressions τ_1, τ_2. It is almost same as proofs of ordinary unification algorithms. What we should note here is the case for the unification between a dynamic type variable and an expression, $\mathit{unify}(\delta, \tau)$.

If $\delta \in FV(\tau)$, then $U(\delta, \tau)$ always returns $U = D \circ \{\delta \Rightarrow D(\tau)\}$ where $D = \mathit{dynamize}(\tau)$, and actually by the same discussion in the previous proof, we have:

$$U = D + \{\delta \Rightarrow D(\tau)\}$$

and we can easily verify that it actually unifies them, $U(\delta) = U(\tau) = D(\tau)$.

Suppose that there is another unifier S such that $S(\delta) = S(\tau)$. Let

$$\begin{array}{l} S_1 = \{(\nu_i \Rightarrow \tau_i) \in S \mid \nu_i \text{ is neither } \delta,\ \alpha_i \text{ nor } \delta_i, \text{ where } (\alpha_i \Rightarrow \delta_i) \in D\} \\ S_2 = \{(\delta_i \Rightarrow S(\alpha_i) \mid \alpha_i \in \mathit{Dom}(D)\} \end{array}$$

$\mathit{Dom}(S_1)$ and $\mathit{Dom}(S_2)$ are distinct, therefore $S' = S_1 + S_2$ is well defined substitution. By using S', S can be written $S = S' \circ U$, because if we let $(\alpha_i \Rightarrow \delta_i) \in D$, then the related type variables δ and α_i in τ are mapped as follows:

$$\begin{array}{l} (S' \circ U)(\alpha_i) = S'(\delta_i) = S_2(\delta_i) = S(\alpha_i) \\ (S' \circ U)(\delta) = S'(D(\tau)) = (S_2 \circ D)(\tau) = S(\tau) = S(\delta) \end{array}$$

If τ is not δ but contains the type variable δ, δ and τ are not unifiable. To prove it, we define some appropriate size of types and suppose there is some unifier S, then compare the size of $S(\tau)$ and $S(\delta)$. Because $\tau \neq \delta$, they cannot be equal and leads a contradiction. In such case, *unify* fails. □

2.6.3 Type Reconstruction algorithm

The type reconstruction algorithm W is in Figure 2.3. $W(\Gamma, e)$ receives a type environment Γ and a bare GML expression e and usually returns a fully type-annotated expression e_t and a substitution. It fails and reports a type error, if its one of queries of identifiers $\mathtt{x}$ in Γ, $\Gamma(\mathtt{x})$ is not defined, or one of the internal unifications fails.

Since all the things related with our extensional polymorphism are hidden in the algorithms *inst* and *unify*, the reconstruction algorithm itself looks very similar to the ordinary ML type reconstruction algorithms by Damas and Milner[DM82] and its variants.

Now we prove the correctness and completeness of the algorithm:

Theorem 9 (Type reconstruction correctness) *The algorithm W is correct with respect to the typing rules. Let Γ be a typing environment and e be a bare GML expression and if $W(\Gamma, e) = e_t,\ S$, then $S(\Gamma) \vdash e_t$.*

PROOF: By induction over the structure of bare GML expression.

Constants Because $\mathit{inst}(\sigma) \leq \sigma$, by using (t-cnst), we have $\Gamma \vdash c_\sigma : \mathit{inst}(\sigma)$.

$$
\begin{array}{rcl}
W(\Gamma,\ c_\sigma) & = & c_\sigma : \mathit{inst}(\sigma),\ \emptyset \\
W(\Gamma,\ \mathtt{g}_\sigma) & = & \mathtt{g}_\sigma : \mathit{inst}(\sigma),\ \emptyset \\
W(\Gamma,\ \mathtt{x}) & = & \mathtt{x}_{\Gamma(\mathtt{x})} : \mathit{inst}(\Gamma(\mathtt{x})),\ \emptyset \\
W(\Gamma,\ \mathtt{fun\ x} \rightarrow e) & = & \text{let } \alpha \text{ be a fresh variable} \\
& & \text{let } e_t : \tau,\ S = W(\Gamma\{\mathtt{x} : \alpha\},\ e) \\
& & \text{in } \mathtt{fun\ x} \rightarrow e_t : S(\alpha) \rightarrow \tau,\ S \\
W(\Gamma,\ e_1\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
& & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\
& & \text{let } \alpha \text{ be a fresh variable} \\
& & \text{let } S = \mathit{unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha) \\
& & \text{in } (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1 \\
W(\Gamma, (e_1\mathtt{,}e_2)) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
& & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\
& & \text{in } S_2(e_{t1})\mathtt{,}e_{t2} : S_2(\tau_1) * \tau_2,\ S_2 \circ S_1 \\
W(\Gamma, \mathtt{let\ x} \mathtt{=} e_1\ \mathtt{in}\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
& & \text{let } \sigma = \mathit{Gen}(S_1(\Gamma), (e_{t1} : \tau_1)) \\
& & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma)\{\mathtt{x} : \sigma\},\ e_2) \\
& & \text{in } \mathtt{let\ x}_{BV(\sigma)} \mathtt{=} S_2(e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2,\ S_2 \circ S_1 \\
\multicolumn{3}{l}{W(\Gamma, \mathtt{let\ rec\ f} \mathtt{=} \mathtt{fun\ x} \rightarrow e_1\ \mathtt{in}\ e_2) =} \\
& & \text{let } \alpha \text{ be a fresh variable} \\
& & \text{let } (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1,\ S_1 = W(\Gamma\{\mathtt{f} : \alpha\},\ \mathtt{fun\ x} \rightarrow e_1) \\
& & \text{let } S_1' = \mathit{unify}(\tau_1, S_1(\alpha)) \\
& & \text{let } \sigma = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)) \\
& & \text{let } e_{t2} : \tau_2,\ S_2 = W(((S_1' \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\},\ e_2) \\
& & \text{in} \\
& & \quad \mathtt{let\ rec\ f}_{BV(\sigma)} \mathtt{=} \\
& & \qquad \{\mathtt{f}_{S_2 \circ S_1'(\tau_1)} \Rightarrow \mathtt{f}_{S_2(\sigma)}\}((S_2 \circ S_1')(\mathtt{fun\ x} \rightarrow e_{t1})) \\
& & \quad \mathtt{in}\ e_{t2} : \tau_2,\ S_2 \circ S_1' \circ S_1
\end{array}
$$

Figure 2.3: Type reconstruction algorithm

Generic primitives Similar to constants, using (t-gen) instead.

Variables Let $\sigma = \Gamma(\mathtt{x})$. Because $inst(\sigma) \leq \sigma$, using (t-inst) we have $\Gamma \vdash \mathtt{x}_\sigma : inst(\sigma)$.

Functions Let

$$W(\Gamma, \mathtt{fun}\ \mathtt{x} \rightarrow e) = \mathtt{fun}\ \mathtt{x} \rightarrow (e_t : \tau) : S(\alpha) \rightarrow \tau,\ S$$
$$\text{where } e_t : \tau,\ S = W(\Gamma\{\mathtt{x} : \alpha\},\ e)$$

By using the induction hypothesis for the reconstruction of e, we have

$$S(\Gamma\{\mathtt{x} : \alpha\}) \vdash e_t : \tau$$

and it equals to $S(\Gamma)\{\mathtt{x} : S(\alpha)\} \vdash e_t : \tau$. Using (t-fun) against this judgment, we obtain

$$S(\Gamma) \vdash \mathtt{fun}\ \mathtt{x} \rightarrow (e_t : \tau) : S(\alpha) \rightarrow \tau$$

Applications Suppose that we have the following:

$$\begin{array}{lcl} W(\Gamma,\ e_1\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\ & & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\ & & \text{let } \alpha \text{ be a fresh variable} \\ & & \text{let } S = \mathit{unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha) \\ & & \text{in } (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1 \end{array}$$

By using the induction hypothesis for the reconstruction of e_1 and e_2, we have:

$$S_1(\Gamma) \vdash e_{t1} : \tau_1$$
$$S_2(S_1(\Gamma)) \vdash e_{t2} : \tau_2$$

Using Theorem 3 against the first judgment with $S \circ S_2$ and the second with S, we obtain:

$$(S \circ S_2)(S_1(\Gamma)) \vdash (S \circ S_2)(e_{t1} : \tau_1)$$
$$S(S_2(S_1(\Gamma))) \vdash S(e_{t2} : \tau_2)$$

Because $(S \circ S_2)(\tau_1) = S(\tau_2 \rightarrow \alpha)$ we can use (t-app):

$$(S \circ S_2 \circ S_1)(\Gamma) \vdash (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha)$$

Tuples It is proven like the applications and simpler than it because of the lack of the unification.

Recursive let definitions We have

$$\begin{array}{l} W(\Gamma, \texttt{let rec f = fun x} \rightarrow e_1 \texttt{ in } e_2) = \\ \quad \text{let } \alpha \text{ be a fresh variable} \\ \quad \text{let } (\texttt{fun x} \rightarrow e_{t1}) : \tau_1,\ S_1 = W(\Gamma\{\texttt{f} : \alpha\},\ \texttt{fun x} \rightarrow e_1) \\ \quad \text{let } S_1' = \mathit{unify}(\tau_1, S_1(\alpha)) \\ \quad \text{let } \sigma = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1)) \\ \quad \text{let } e_{t2} : \tau_2,\ S_2 = W(((S_1' \circ S_1)(\Gamma))\{\texttt{f} : \sigma\},\ e_2) \\ \quad \text{in } \ \texttt{let rec f}_{BV(\sigma)} \texttt{=} \{\texttt{f}_{S_2 \circ S_1'(\tau_1)} \Rightarrow \texttt{f}_{S_2(\sigma)}\}((S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1})) \\ \quad \quad \texttt{in } e_{t2} : \tau_2,\ S_2 \circ S_1' \circ S_1 \end{array}$$

Using the induction hypothesis for the reconstruction of $\texttt{fun x} \rightarrow e_1$

$$S_1(\Gamma\{\texttt{f} : \alpha\}) \vdash \texttt{fun x} \rightarrow e_{t1} : \tau_1$$

By applying Theorem 3 with $S_2 \circ S_1'$,

$$(S_2 \circ S_1' \circ S_1)(\Gamma\{\texttt{f} : \alpha\}) \vdash (S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1} : \tau_1)$$

On the other hand, using the induction hypothesis for the reconstruction of e_2,

$$S_2((S_1' \circ S_1)(\Gamma))\{\texttt{f} : S_2(\sigma)\} \vdash e_{t2} : \tau_2$$

From the definition of $\sigma = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1))$, by the same argument done in the `let` part of the proof of Theorem 3,

$$\begin{array}{rcl} S_2(\sigma) & = & S_2(\mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1))) \\ & = & \mathit{Gen}((S_2 \circ S_1' \circ S_1)(\Gamma), (S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1} : \tau_1)) \end{array}$$

These two judgment and the generalization, we can construct the goal using (t-letrec):

$$\begin{array}{ll} (S_2 \circ S_1' \circ S_1)(\Gamma) \vdash & \texttt{let rec f}_{BV(\sigma)} \texttt{=} \{\texttt{f}_{S_2 \circ S_1'(\tau_1)} \Rightarrow \texttt{f}_{S_2(\sigma)}\}(\texttt{fun x} \rightarrow (S_2 \circ S_1')(e_{t1})) \\ & \texttt{in } e_{t2} : \tau_2 \end{array}$$

Let definitions Similar to the recursive lets, and easier than them.

□

The algorithm W is not only correct, but also *complete*. If given expression has some typing, it always returns the most general *principal type*:

Theorem 10 (Type reconstruction completeness) *The algorithm W is complete. Let Γ be a typing environment and e_t' be a GML expression. If there exists a typing $S'(\Gamma) \vdash e_t' : \tau'$ where $\mathit{strip}(e_t') = e$ and $\mathit{Dom}(S') \subset FV(\Gamma)$, then $W(\Gamma, e) = (e_t : \tau),\ S$ and there exists a substitution S'' such that*

$$S''(\tau) = \tau', \quad \forall \nu \notin \mathit{fresh}(W(\Gamma, e)), \quad S'(\nu) = S'' \circ S(\nu)$$

In the latter statement, $\mathit{fresh}(W(\ldots))$ denotes the set of newly introduced type variables in the algorithm W.

PROOF: Like the proof of the previous theorem, by induction over the structure of the bare GML expression.

Variables Let $e'_t = \mathtt{x} : \tau'$. Because $S'(\Gamma) \vdash \mathtt{x} : \tau'$, we have a binding of $\mathtt{x}$ in Γ. We let $\Gamma(\mathtt{x}) = \sigma$. Then from the typing, $\tau' \leq S'(\sigma)$.

Since Γ has a binding for $\mathtt{x}$, the reconstruction does not fail:

$$W(\Gamma, \mathit{strip}(e'_t)) = W(\Gamma, \mathtt{x}) = \mathtt{x} : \mathit{inst}(\sigma),\ \emptyset$$

If we let $\tau = \mathit{inst}(\sigma)$, $e_t = \mathtt{x} : \tau$ and $S = \emptyset$. Let $\sigma = \forall \nu_1 \ldots \nu_n.\tau_0$. By renaming the bound variables, we can suppose that $\{\nu_i\}$ are out of the reach of S' and $\nu_i \notin FV(\Gamma) \cup FV(\tau')$. Then there exists fresh type variables ν'_i and τ is written as:

$$\tau = \{\nu_i \Rightarrow \nu'_i\}(\tau_0)$$

Therefore $\tau_0 = \{\nu'_i \Rightarrow \nu_i\}(\tau)$. On the other hand, since $S'(\sigma) = \forall \nu_1 \ldots \nu_n.S'(\tau_0)$, τ' can be written as

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ S')(\tau_0)$$

for some types $\tau_1, \ldots, \tau_n$. From these,

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ S' \circ \{\nu'_i \Rightarrow \nu_i\})(\tau)$$

Since $\{\nu_i\}$ and $\{\nu'_i\}$ are both out of the reach of S', this becomes

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ \{\nu'_i \Rightarrow \nu_i\} \circ S')(\tau) = (\{\nu'_i \Rightarrow \tau_i\} \circ S')(\tau)$$

Let $S'' = \{\nu'_i \Rightarrow \tau_i\} \circ S'$, then

$$S''(e_t) = \mathtt{x} : S''(\tau) = \mathtt{x} : \tau' = e'_t$$

For any type variable ν except ν'_i,

$$S'' \circ S(\nu) = S''(\nu) = \{\nu'_i \Rightarrow \tau_i\}(S'(\nu)) = S'(\nu)$$

since $S = \emptyset$ and ν'_i are fresh.

Constants, Generic primitives They are proven more simply than the cases for variables.

Functions The typing $S'(\Gamma) \vdash \texttt{fun x} \to e_1 : \tau'$. Then the typing derivation must end like:

$$\frac{S'(\Gamma)\{\texttt{x} = \tau'_1\} \vdash e'_{t1} : \tau'_2}{S'(\Gamma) \vdash \texttt{fun x} \to e'_{t1} : \tau'_1 \to \tau'_2}$$

where $\tau' = \tau'_1 \to \tau'_2$ and $strip(e'_{t1}) = e_1$.

Let α be a fresh variable, $\Gamma_1 = \Gamma\{\texttt{x} = \alpha\}$ and $S'_1 = S'\{\alpha \Rightarrow \tau'_1\}$, then $S'_1(\Gamma_1) = S'(\Gamma)\{\texttt{x} = \tau'_1\}$. Therefore the premise is rewritten as $S'_1(\Gamma_1) \vdash e'_{t1} : \tau'_2$. Applying the induction hypothesis to this typing, we obtain

$$W(\Gamma_1, e_1) = e_{t1} : \tau_2,\ S_1$$

and there is a substitution S''_1 such that

$$S''_1(e_{t1} : \tau_2) = e'_{t1} : \tau'_2,\quad S'_1(\nu) = S''_1 \circ S_1(\nu) \text{ for any } \nu \notin \mathit{fresh}(W(\Gamma_1, e_1))$$

The latter part is rewritten into the followings:

$$\begin{array}{l} S'_1(\alpha) = \tau'_1 = (S''_1 \circ S_1)(\alpha) \\ \forall \nu \notin \mathit{fresh}(W(\Gamma_1, e_1)) \cup \{\alpha\},\quad S'_1(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Because $W(\Gamma_1, e_1)$ is well-defined, $W(\Gamma, \texttt{fun x} \to e_1)$ is also well-defined:

$$W(\Gamma, \texttt{fun x} \to e_1) = \texttt{fun x} \to e_t :\ S_1(\alpha) \to \tau_2,\ S_1$$

The fresh variables introduced by $W(\Gamma, \texttt{fun x} \to e_1)$ are α and those of $W(\Gamma_1, e_1)$, therefore, S''_1 has the following properties, because $\mathit{fresh}(W(\Gamma, \texttt{fun x} \to e_1)) = \mathit{fresh}(W(\Gamma_1, e_1)) \setminus \{\alpha\}$.

$$\begin{array}{l} S''_1(S_1(\alpha)) \to \tau'_2 = \tau'_1 \to \tau'_2 \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, \texttt{fun x} \to e_1)),\quad S'_1(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Applications Let $e'_t = e'_{t1}\ e'_{t2}$. The typing ends like

$$\frac{S'(\Gamma) \vdash e'_{t1} : \tau'_1 \to \tau'_2 \qquad S'(\Gamma) \vdash e'_{t2} : \tau'_1}{S'(\Gamma) \vdash e'_{t1}\ e'_{t2} : \tau'_2}$$

where $strip(e'_{ti}) = e_i$.

Applying the induction hypothesis to e'_{t1}, we have

$$W(\Gamma, strip(e'_{t1})) = e_{t1} : \tau_1,\ S_1$$

and there exists a substitution S''_1 such that

$$\begin{array}{l} S''_1(e_{t1} : \tau_1) = e'_{t1} : \tau'_1 \to \tau'_2 \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, strip(e'_{t1}))),\quad S'(\nu) = S''_1 \circ S_1(\nu) \end{array}$$

$S'(\Gamma) = S_1''(S_1(\Gamma))$, since Γ never contains the fresh type variables introduced by $W(\Gamma, \mathit{strip}(e'_{t1}))$.

Using the induction hypothesis for the typing of e'_{t2}, $S_1''(S_1(\Gamma)) \vdash e'_{t2} : \tau_1'$:

$$W(S_1(\Gamma), \mathit{strip}(e'_{t2})) = e_{t2} : \tau_2,\ S_2$$

and there exists S_2'' such that

$$S_2''(e_{t2} : \tau_2) = e'_{t2} : \tau_1',\ \forall \nu \in \mathit{fresh}(W(S_1(\Gamma), \mathit{strip}(e'_{t2}))),\quad S_1''(\nu) = S_2'' \circ S_2(\nu)$$

Adding a binding $(\alpha \Rightarrow \tau_2')$ of a fresh type variable α to this S_2'', the substitution $S_2''\{\alpha \Rightarrow \tau_2'\}$ unifies $S_2(\tau_1)$ and $\tau_2 \to \alpha$ as follows, since $FV(\tau_1) \cap \mathit{fresh}(W(S_1(\Gamma), \mathit{strip}(e'_{t2})) = \emptyset$ and α is fresh:

$$\begin{array}{l} S_2''\{\alpha \Rightarrow \tau_2'\}(S_2(\tau_1)) = S_2''(S_2(\tau_1)) = S_1''(\tau_1) = \tau_1' \to \tau_2' \\ S_2''\{\alpha \Rightarrow \tau_2'\}(\tau_2 \to \alpha) = S_2''(\tau_2) \to \tau_2' = \tau_1' \to \tau_2' \end{array}$$

From the existence of this unifier $S_2''\{\alpha \Rightarrow \tau_2'\}$ assures the existence of the most general unifier of these types $S = \mathit{unify}(S_2(\tau_1), \tau_2 \to \alpha)$. Therefore $W(\Gamma, e_1\ e_2)$ is well defined:

$$W(\Gamma, e_1\ e_2) = (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1$$

Because S is a more general unifier than $S_2''\{\alpha \Rightarrow \tau_2'\}$, there must be a substitution S'' such that

$$S_2''\{\alpha \Rightarrow \tau_2'\} = S'' \circ S$$

This S'' is what we want:

$$\begin{array}{l} S''(S(\alpha)) = S_2''\{\alpha \Rightarrow \tau_2'\}(\alpha) = \tau_2' \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, e_1\ e_2)) = \mathit{fresh}(W(\Gamma, e_1)) \cup \mathit{fresh}(W(S_1(\Gamma), e_2)) \cup \{\alpha\}, \end{array}$$
$$\begin{array}{rcll} (S'' \circ S \circ S_2 \circ S_1)(\nu) & = & (S_2''\{\alpha \Rightarrow \tau_2'\}) \circ S_2 \circ S_1(\nu) & \text{from } S'' \circ S = S_2''\{\alpha \Rightarrow \tau_2'\} \\ & = & S_2'' \circ S_2 \circ S_1(\nu) & \text{because } \alpha \text{ is fresh} \\ & = & S_1'' \circ S_1(\nu) & \text{from } \nu \notin \mathit{fresh}(W(S_1(\Gamma), e_2)) \\ & = & S'(\nu) & \text{from } \nu \notin \mathit{fresh}(W(\Gamma, e_1)) \end{array}$$

Tuples Like applications, but simpler due to the lack of the unification.

Recursive let The typing is written as $S'(\Gamma) \vdash \texttt{let f = fun x} \to e'_{t1}\ \texttt{in}\ e'_{t2} : \tau'$ where $\mathit{strip}(e'_{ti}) = e_i$ and its type derivation must end with the following derivation:

$$\frac{\begin{array}{c} S'(\Gamma)\{\texttt{f} : \tau_1'\} \vdash \texttt{fun x} \to e'_{t1} : \tau_1' \quad \forall V'.\tau_1' = \mathit{Gen}(S'(\Gamma), (\texttt{fun x} \to e'_{t1}) : \tau_1') \\ S'(\Gamma)\{\texttt{f} : \forall V'.\tau_1'\} \vdash e'_{t2} : \tau_2' \end{array}}{S'(\Gamma) \vdash \texttt{let rec f}_{V'} \texttt{=} \{\texttt{f}_{\tau_1'} \Rightarrow \texttt{f}_{\forall V'.\tau_1'}\}(\texttt{fun x} \to e'_{t1})\ \texttt{in}\ e'_{t2} : \tau_2'}$$

where $strip(e'_{t1}) = e_1$ and $strip(e'_{t2}) = e_2$. Let α be a fresh variable. Then, like what we did in the part of the functions,

$$S'\{\alpha \Rightarrow \tau'_1\}(\Gamma\{\mathtt{f} : \alpha\}) = S'(\Gamma)\{\mathtt{f} : \tau'_1\}$$

Therefore we can use the induction hypothesis for the reconstruction of $\mathtt{fun\ x} \rightarrow e_1$:

$$W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1) = (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1,\ S_1$$

and there is a substitution S''_1 such that

$$S''_1(\tau_1) = \tau'_1, \quad \forall \nu \notin \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1)),\ S'\{\alpha \Rightarrow \tau'_1\}(\nu) = (S''_1 \circ S_1)(\nu)$$

The latter part is rewritten into the followings:

$$\begin{array}{l} S'\{\alpha \Rightarrow \tau'_1\}(\alpha) = \tau'_1 = (S''_1 \circ S_1)(\alpha) \\ \forall \nu \notin \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e)) \cup \{\alpha\},\ S'\{\alpha \Rightarrow \tau'_1\}(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Because $S''_1(\tau_1) = \tau'_1 = S''_1(S_1(\alpha))$, S''_1 is a unifier of τ_1 and $S_1(\alpha)$. This implies the existence of the most general unifier S'_1 of them:

$$S'_1 = \mathit{unify}(\tau_1, S_1(\alpha))$$

and there exists a substitution S'''_1 such that $S''_1 = S'''_1 \circ S'_1$.

Let the type scheme obtained from the reconstruction σ. This and the scheme from the derivation are:

$$\begin{array}{rcl} \sigma & = & \forall V.S'_1(\tau_1) \\ & = & \mathit{Gen}((S'_1 \circ S_1)(\Gamma), S'_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \\ \\ \forall V'.\tau'_1 & = & \mathit{Gen}(S'(\Gamma), (\mathtt{fun\ x} \rightarrow e'_{t1}) : \tau'_1) \\ & = & \mathit{Gen}((S''_1 \circ S_1)(\Gamma), S''_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \\ & = & \mathit{Gen}((S'''_1 \circ S'_1 \circ S_1)(\Gamma), S'''_1 \circ S'_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \end{array}$$

From Lemma 5, $\forall V'.\tau'_1 \leq \sigma$.

Using Lemma 6, from the last premise, we obtain a typing $S'(\Gamma)\{\mathtt{f} : \sigma\} \vdash e''_{t2} : \tau'_2$, where $strip(e''_{t2}) = strip(e'_{t2})$.

$$\begin{array}{rcll} S'(\Gamma)\{\mathtt{f} : \sigma\} & = & (S'\{\alpha \Rightarrow \tau'_1\}(\Gamma))\{\mathtt{f} : \sigma\} & \alpha \text{ never appears in } \Gamma \\ & = & ((S''_1 \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\} & \Gamma \text{ contains no fresh variables} \\ & = & ((S'''_1 \circ S'_1 \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\} & \\ & = & ((S'''_1 \circ S'_1 \circ S_1)(\Gamma))\{\mathtt{f} : S'''_1(\sigma)\} & \\ & = & S'''_1((S'_1 \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}) & \end{array}$$

The typing is now rewritten as follows:

$$S'''_1((S'_1 \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}) \vdash e''_{t2} : \tau'_2$$

From it we can use the induction hypothesis for the reconstruction of e_2:

$$W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2) = e_{t2} : \tau_2,\ S_2$$

and there exists a substitution S_2'' such that

$$S_2''(\tau_2) = \tau_2', \quad \forall \nu \notin \mathit{fresh}(W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)),\ S_1'''(\nu) = (S_2'' \circ S_2)(\nu)$$

The well-foundness of $W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)$ assures that the whole reconstruction $W(\Gamma, \mathtt{let\ rec\ f = fun\ x} \rightarrow e_1 \mathtt{\ in\ } e_2)$ is also well defined. S_2'' has also another property, which we want:

$$\begin{array}{l} \forall \nu \notin \mathit{fresh}(W(\Gamma, \mathtt{let\ rec\ f} = e_1 \mathtt{\ in\ } e_2)) = \\ \qquad \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1)) \cup \mathit{fresh}(W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)) \cup \{\alpha\}, \\ \begin{array}{rcl} (S_2'' \circ (S_2 \circ S_1' \circ S_1))(\nu) & = & (S_1''' \circ S_1' \circ S_1)(\nu) \\ & = & (S_1'' \circ S_1)(\nu) \\ & = & S'(\nu) \end{array} \end{array}$$

Non recursive let Provable more simply than the recursive let bindings.

□

2.7 Conclusion

In this chapter, we formalized the first version of our target language GML, an extension of ML which supports basic mechanisms of extensional polymorphism: generic values, ad-hoc polymorphic values whose semantics are dependent on types at evaluation.

First, we introduced dynamic type variables, a new set of type variables to the type system. Having quantified dynamic type variables in type schemes, generic values are distinguished from the other normal values in the type level. The distinction between generic values and the others in type schemes plays an important role to apply special rules of evaluation and typing to generic values. In this chapter, generic values are provided as primitives or their derived polymorphic values. User construction of generic values will be argued later in Chapter 7.

To achieve the type-dependent evaluation, the semantics of GML are defined as an operation semantics system over fully type-annotated GML expressions. When an instance of a polymorphic value is evaluated, its type instantiation is applied to the corresponding value stored in the evaluation environment, so that it can be evaluated under the correct evaluation type context. Generic values are evaluated to functions which take type contexts and return appropriate semantic values of the corresponding type contexts.

A type system was built to restrict GML programs to well-typed expressions which are never evaluated to run time errors. The type system is also in the ML style, but it is defined over the fully type-annotated expressions, as the semantics system. We also defined a type reconstruction algorithm which type annotates bare GML expressions. This algorithm is complete in respect with the typing rules: if a bare expression is typable, it always returns the principal type annotation. The semantics of a bare expression was defined as those under the corresponding principal typing.

Evaluation of expressions with fully type annotations is more complicated than the classical ML semantics without types. Direct implementation of the GML semantics with type instantiation cannot compete with existing implementations of classical semantics. Therefore in the next chapter, we provide a compilation method of GML expressions to an untyped ML language, whose semantics is type free. The type instantiation of GML evaluation rules is translated to *run time type dispatching*, delivering of values which represent ML types to generic values. Since our compilation performs this run time type dispatching only for generic instances, efficiency of the other non generic part of programs are run time type free, hence it can be as efficiently as the classical ML evaluation.