Chapter 3

Compilation of GML

3.1 Introduction

Compilation with run time types

In normal ML, a type soundness theorem can remove all the type information from evaluation systems, and programs are executed efficiently without run time type handling. In GML evaluation, thanks to its type soundness (Theorem 4) we need not to implement error handling either. But the evaluation system itself cannot be completely free from type manipulations, since semantics of generic instances require their evaluation type context. Such semantics with types is difficult to implement directly and is inefficient. Instead, we here compile GML expressions to an intermediate ML language without types. The type instantiations in the GML semantics are transformed to *run time type dispatching* using type abstractions and applications as the second order typed lambda calculus and system F[GTL90]. The type instantiation information is delivered as *run time type* functional arguments, value representations of type expressions. Generic primitives and their derived generic values are compiled to functions which take these run time types to obtain appropriate evaluation type contexts. Since the run time type dispatching is performed only against generic instances, the other normal non-generic part can be evaluated as efficiently as in normal ML implementations.

Outline of this chapter

We first examine generic related semantic evaluation rules, and argue informally our strategy of compilation in Section 3.2. Then in Section 3.3, we define our intermediate language for compilation, IML, and its semantics. In Section 3.4, we define our compilation algorithm which translates fully type-annotated GML expressions to IML. Finally in Section 3.5, we will check both semantics of GML and IML and and prove our compilation is correct. At the end of this chapter, we argue about the coherence problem of implicitly typed programs.

3.2 Analysis of the generic evaluation

GML evaluation rules for generic values contain type related operations. We here look at them closer again and try to figure out what they really do, and grasp some rough sketch of our compilation strategy:

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

An instance of a generic primitive $\mathtt{g}_\sigma : \tau$ returns a semantic value $gen(\mathtt{g}, \tau)$ which depends on its evaluation type context τ. If $\mathtt{g}$ is used in a derived generic definition, evaluation type contexts of $\mathtt{g}$ are not the type annotation statically obtained at typing itself, but a type expression obtained by applying a substitution which is delivered by the following two evaluation rules:

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

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

If a derived generic value is defined by (e-g-let) or (e-g-let-r), its definition code e_t is stored as a generic closure $gclos(E, *, V, e_t)$, and it is evaluated when the derived generic value is actually used. When an instance of such a derived generic value is evaluated in a type context τ, (e-gvar) or (e-gvar-r) is used for the evaluation. It applies the instantiation S to the code in the generic closure e_t, then evaluate it, where S indicates how the generic value is instantiated at evaluation. This instantiation application provides evaluation type contexts for generic primitives or derived generic values appear inside e_t.

Clearly, (e-g-let$*$) are only the rules which introduce generic closures into evaluation environment, and they are consumed only by (e-gvar$*$) rules. In (e-gvar$*$), an instantiation S is applied to all the type annotations of the code e_t, so that the inside generic primitives can receive instantiations of their evaluation contexts. However, on the other hand, this type substitution takes no semantical effect on the other non-generic stuffs, since the other evaluation rules than the above rules never use type annotations in any essential way. In addition, since the type schemes of generic primitives have no regular variables, bindings of regular variables in an instantiation S play no role in determining the semantics of generic primitives.

It leads to the following ideas of our compilation.

Non-generic values

At evaluation all the type annotations of non- generic expressions can be ignored since their evaluation is independent of their type annotations. This means that we can use the same compilation method as normal ML language for the non-generic part of our language. This also assures that as far as the non-generic part concerns, compilation will be as efficiently as normal ML compilation without any overhead.

Generic closures

A derived generic value stored as a generic closure in an evaluation environment waits information how its quantified dynamic type variables are actually instantiated for each instance. Then it sends the type instantiation information to the internal generic values in its definition code. This mechanism can be replaced by a function – a closure which takes arguments of instantiation information and send them to the internal compilation of generic instances by function application. For example, if there is a generic closure $gclos(E, \_, V, e_t)$ whose generalized type variable set contain n dynamic type variables, i.e.

$$V \cap \mathbb{D} = \{\delta_1, \ldots \delta_n\}$$

then creating a new corresponding identifier $\mathtt{d}_i$ for each δ_i, the generic closure is transformed to a normal closure who takes these $\mathtt{d}_i$:

$$\begin{array}{ccc} \text{GML semantics} & & \text{Compilation semantics} \\ gclos(E, \_, V, e_t) & \Longrightarrow & clos(\mathcal{E}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2\ \rightarrow\ \ldots\ \rightarrow \mathtt{fun}\ \mathtt{d}_n \rightarrow e) \end{array}$$

Here, the evaluation environments $\mathcal{E}$ and the code e in the right hand side should be some compilation of E and e_t.

Because the generic closure are introduced only by (e-g-let$*$) rules, this semantic translation can be ported into the syntactic compilation as rewriting of let expressions without much difficulty. The let binding which stores the above generic closure $gclos(E, \_, V, e_t)$ can be written as $\mathtt{let}\ \mathtt{x}_V = (e_t : \tau_1)\ \mathtt{in}\ \ldots$ and this GML expression is compiled to the following code, whose semantics is equal to the above semantics compilation of the generic closure:

$$\begin{array}{ccll} \text{GML expression} & & \text{Compiled expression} & \\ \mathtt{let}\ \mathtt{x}_V = (e_t : \tau_1)\ \mathtt{in}\ \ldots & \Longrightarrow & \mathtt{let}\ \mathtt{x} = & \mathtt{fun}\ \mathtt{d}_1 \rightarrow \\ & & & \ldots \\ & & & \mathtt{fun}\ \mathtt{d}_n \rightarrow e\ \ \mathtt{in}\ \ldots \end{array}$$

Generic instances

When a generic instance $\mathtt{x}$ is evaluated, there must be a binding of this identifier to a generic closure $gclos(E, \_, V, e_t)$ inside the current evaluation environment. In our compilation, this generic closure will be translated into a sequence of normal closures as we described above. The evaluation of the generic closure code with its type annotation substitution $S(e_t)$ in GML will be simulated by a sequence of applications of type informations to these newly introduced normal closures:

GML evaluation

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

$$\Downarrow$$

Evaluation of compilation

$$\frac{\dfrac{\mathcal{E}\{\mathtt{d}_1 = «S(\delta_1)», \dots \mathtt{d}_n = «S(\delta_n)»\} \models e \Longrightarrow r}{\dots}}{\mathcal{E}'\{\mathtt{x} = clos(\mathcal{E}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \rightarrow\ \dots\ \rightarrow \mathtt{d}_n \rightarrow e)\} \models \mathtt{x}\ «S(\delta_1)»\ \dots\ «S(\delta_n)» \Longrightarrow r}$$

The application $\mathtt{x}\ «S(\delta_1)»\ \dots\ «S(\delta_n)»$ binds the variables $\mathtt{d}_i$ by *run time type representations* of $S(\delta_i)$, $«S(\delta_i)»$ and evaluate e, the code in the closure under the updated environment. The run time type is a predefined data type which is isomorphic to the type expressions. Its definition will be given later formally. Each run time type $«S(\delta_i)»$ denotes how the generalized dynamic type variable δ_i in the generalized type variable set V is instantiated by the instantiation S.

We have to be careful about the orders of these run time type arguments and the λ-abstractions for compilations of generic closures so that instantiation information of a generalized dynamic type variable is correctly passed to its corresponding abstraction in compilation. We need to have the uniquely ordered list of quantified type variables in a type scheme $\forall V.\tau$. The following function *Vlist* computes such a list ('@' is the list concatenation operator here):

$$\begin{array}{rcl}
\mathit{Vlist}(V, \mathit{base}) & = & [\,] \\
\mathit{Vlist}(V, \nu) & = & [\nu] \text{ if } \nu \in V \\
 & & [\,] \text{ otherwise} \\
\mathit{Vlist}(V, \tau_1 \rightarrow \tau_2) & = & \text{let } l = \mathit{Vlist}(V, \tau_1) \\
 & & \text{let } V' = \{\nu \in V \mid \nu \notin l\} \text{ in} \\
 & & l\,@\,\mathit{Vlist}(V', \tau_2) \\
\mathit{Vlist}(V, \tau_1 * \tau_2) & = & \text{let } l = \mathit{Vlist}(V, \tau_1) \\
 & & \text{let } V' = \{\nu \in V \mid \nu \notin l\} \text{ in} \\
 & & l\,@\,\mathit{Vlist}(V', \tau_2)
\end{array}$$

The function $\mathit{Vlist}(V, \tau)$ lists the type variables in V in the order of their first occurrences in τ. For example,

$$\mathit{Vlist}(\{\alpha, \delta\}, (\delta * \alpha) \rightarrow (\delta * \beta)) = [\delta;\ \alpha]$$

In compilation of generic closures and instances, we use this ordering function *Vlist* and obtain the unique ordered list of their generalized dynamic type variables:

$$[\delta_1;\ \dots;\ \delta_n] = \mathit{Vlist}(V \cap \mathbb{D}, \tau)$$

Abstractions added for compilations of derived let definitions and run time types arguments applied to them in compilations for generic instances must be arranged so that their corresponding type variables are in this order.

Generic primitives

Thus instantiation information of derived generic values are passed into their definition codes as run time types. The final destinations of these informations are generic primitive instances $\mathsf{g}_\sigma : \tau$, where τ is the type annotation approved by type system. When this instance g_σ is evaluated by (g-var), it should be instantiated by some instantiation S and becomes $S(\tau)$. In the compilation, instead of applying this substitution, the evaluation type context will be obtained as a form of function application of run time types, where each run time type denotes how each generalized dynamic type variable of σ is instantiated:

GML evaluation
$$E \models \mathsf{g}_{\forall V.\tau'} : S(\tau) \Longrightarrow \mathit{gen}(\mathsf{g}, S(\tau))$$

$$\Downarrow$$

Evaluation of compilation
$$\frac{[\delta_1; \ldots; \delta_n] = \mathit{Vlist}(V, \tau') \qquad \mathcal{E} \models e_1 \Longrightarrow «\tau_1» \ \ldots \ \mathcal{E} \models e_n \Longrightarrow «\tau_n»}{\mathcal{E} \models \mathsf{g}\ e_1\ \ldots\ e_n \Longrightarrow \mathit{gen}'(\mathsf{g}, \{\delta_i \Rightarrow \tau_i\}(\tau')) = \mathit{gen}'(\mathsf{g}, S(\tau))}$$

In the evaluation of the compilation, the instantiation information of each generalized dynamic type variable δ_i of $\forall V.\tau'$ is given by the expression e_i, which is evaluated to a run time type $«\tau_i»$. We will have a special evaluation rule for generic primitives which collects these run time type arguments and rebuild the evaluation type context $S(\tau) = \{\delta_i \Rightarrow \tau_i\}(\tau')$. Instead of the original generic primitive semantic query function $\mathit{gen}(\mathsf{g}, S(\tau))$, we need to use $\mathit{gen}'(\mathsf{g}, S(\tau))$ which returns a semantic value of the compiled language semantics, which is *compatible* with the original. This compatibility of semantic values between two systems will be defined later.

3.3 The intermediate language

We formalize the ideas in the previous section. The compilation of our language is done in two steps. Rather than compiling directly from a GML program to an executable, first we translate fully type-annotated GML expressions into expressions without type annotations and generic stuffs. They are almost a subset of the bare GML expressions, but to make the discussion clearer, we should define it as *an intermediate ML like language* called IML. The grammar of IML is obtained by removing generic primitives and explicit type annotations from the bare GML, adding run time type representation $RT(e_1, \ldots, e_n)$. For each run time type constructor RT has its specific arity n and always takes n arguments, like ordinary ML data constructors.

$$\begin{array}{rcll}
e & ::= & c_\sigma & \\
& | & RT(e, \ldots, e) & \text{Run time types} \\
& | & \mathtt{x} & \\
& | & \mathtt{fun}\ \mathtt{x} \rightarrow e & \\
& | & e\ e & \\
& | & e, e & \\
& | & \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e & \\
& | & \mathtt{let}\ \mathtt{rec}\ \mathtt{x} = e\ \mathtt{in}\ e &
\end{array}$$

The obtained IML syntax is quite familiar to the ML programmers and can be executed using any existing compilation technique of ML languages.

3.3.1 Run time types

To pass the evaluation type contexts of generic instances at run time, we need to define a new built-in data type, *run time types*. The run time types are isomorphic to the type expressions of GML defined in Section 2.2.1, except that it has no representation for dynamic type variables. The set of the constructors RT and their definitions are as follows:

```
type rtype =
| RTbase   of rpath                              (* basic types *)
| RTvar    of int                             (* type variables *)
| RTarrow  of rtype * rtype                   (* function types *)
| RTtuple  of rtype * rtype                           (* tuples *)
| RTconstr of rpath * rtype list                  (* data types *)
```

Each constructor represents the corresponding type expression construct: `RTbase` for basic types, `RTvar` for type variables, `RTarrow` for functional types and `RTtuple` for tuples. Type variables are expressed by their unique integer identifications. user-defined data types such as variants and records are not included in our type system for simplicity, but their representation in run time types can be given as `RTconstr` with their paths and type arguments. Basic and user-defined data types are identified by their *run time path* `rpath`, run time representation of data type identification. At this moment they are equivalent to their names[1].

In run time types, we have only one kind of type variables and do not need special distinction of regular and dynamic type variables, because all the run time types created by the compilation algorithm will represent type expressions without regular type variables.

Translation from type expressions to run time types is straightforward. The run time representation of a type expression τ, «τ» is defined as follows:

$$\begin{array}{rcl}
\text{«}\mathit{base}\text{»} & = & \texttt{RTbase}\ \text{"}\mathit{base}\text{"} \\
\text{«}\delta_i\text{»} & = & \texttt{RTvar}\ i \\
\text{«}\tau_1 \to \tau_2\text{»} & = & \texttt{RTarrow(}\text{«}\tau_1\text{»}, \text{«}\tau_2\text{»}\texttt{)} \\
\text{«}\tau_1 * \tau_2\text{»} & = & \texttt{RTtuple(}\text{«}\tau_1\text{»}, \text{«}\tau_2\text{»}\texttt{)} \\
\text{«}(\tau_1, \ldots,\ \tau_n)\ t\text{»} & = & \texttt{RTconstr(}\text{"}t\text{"},\ \texttt{[}\text{«}\tau_1\text{»}; \ldots;\ \text{«}\tau_n\text{»}\texttt{])}
\end{array}$$

Each type variable δ is translated to `RTvar` i, where i is the integer index for δ. This index is uniquely defined for each dynamic type variable. We suppose when we write δ_i, its index is always i. Using this integer index, representations of the same type variables share the same index number of `RTvar`. For example,

$$\text{«}(\delta_1 * \delta_2) \to \delta_1\text{»} = \texttt{RTarrow(RTtuple(RTvar 1, RTvar 2), RTvar 1)}$$

[1]They will be extended to have the fingerprints of their definitions in Chapter 5, to exchange these run time types between different programs.

If τ contains regular type variables, $«\tau»$ is not defined. However, all the run time types created by our compilation algorithm will be well defined, because all the run time type creations are done for type expressions which have no regular type variables. If τ has no regular type variable occurrence, since this translation is somewhat canonical, we usually use $«\tau»$ notation for run time types in IML for simplicity.

3.3.2 Semantics of IML

The semantics of IML is much easier than one of GML. It is same as normal ML like language therefore straightforward. The semantic values for IML are obtained by adding those for run time types to the values for GML without type annotations in closures:

$$
\begin{array}{lcll}
r & ::= & v & \text{Values} \\
 & | & err & \text{Errors} \\
v & ::= & b_\sigma & \text{Basic constants} \\
 & | & RT(v,\ldots,v) & \text{Run time types} \\
 & | & v,v & \text{Tuples} \\
 & | & clos(\mathcal{E},\_,\mathtt{x},e) & \text{Functional closures} \\
 & | & clos(\mathcal{E},\mathtt{f},\mathtt{x},e) & \text{Functional closures (recursive)} \\
\mathcal{E} & ::= & \{\mathtt{x}=v,\ \ldots\ \mathtt{x}=v\} & \text{Environment}
\end{array}
$$

Run time type semantic values use the same constructors RT and their arities as those for the IML syntax. We also use the notation $«\tau»$ for the semantic value for a type τ.

An evaluation judgment for IML is written as $\mathcal{E} \models_i e \Longrightarrow v$. The semantic rules are also defined in the same way as GML's, but this time we have no type annotations:

(i-cnst) $\mathcal{E} \models_i c_\sigma \Longrightarrow cnst'(c_\sigma)$

(i-runtype) $$\frac{\mathcal{E} \models_i e_i \Longrightarrow v_i}{\mathcal{E} \models_i RT(e_1,\ldots,e_n) \Longrightarrow RT(v_1,\ldots,v_n)}$$

(i-gen) $$\frac{\sigma = \forall V.\tau \quad [\delta_1;\ldots;\delta_n] = Vlist(V,\tau) \qquad \mathcal{E} \models_i e_1 \Longrightarrow «\tau_1» \ \ldots\ \mathcal{E} \models_i e_n \Longrightarrow «\tau_n»}{E \models_i \mathtt{g}_\sigma\ e_1\ \ldots\ e_n \Longrightarrow gen'(\mathtt{g}, \{\delta_i \Rightarrow \tau_i\}(\tau))}$$

(i-tpl) $$\frac{\mathcal{E} \models_i e_1 \Longrightarrow v_1 \quad \mathcal{E} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i (e_1,e_2) \Longrightarrow v_1,v_2}$$

(i-var) $\mathcal{E}\{\mathtt{x}=v\} \models_i \mathtt{x} \Longrightarrow v$

(i-fun) $\mathcal{E} \models_i \mathtt{fun\ x} \rightarrow e \Longrightarrow clos(E,\_,\mathtt{x},e)$

$$\textbf{(i-let)}\ \frac{\mathcal{E} \models_i e_1 \Longrightarrow v_1 \quad \mathcal{E}\{\mathtt{x} = v_1\} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 \Longrightarrow v_2}$$

$$\textbf{(i-letrec)}\ \frac{\mathcal{E}\{\mathtt{x} = clos(\mathcal{E}, \mathtt{f}, \mathtt{x}, e_1)\} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i \mathtt{let\ rec\ f = fun\ x} \to e_1\ \mathtt{in}\ e_2 \Longrightarrow v_2}$$

$$\textbf{(i-app1)}\ \frac{\begin{array}{c}\mathcal{E} \models_i e_2 \Longrightarrow v_2 \quad \mathcal{E} \models_i e_1 \Longrightarrow clos(\mathcal{E}', \_, \mathtt{x}, e) \\ \mathcal{E}'\{\mathtt{x} = v_2\} \models_i e_t \Longrightarrow v\end{array}}{\mathcal{E} \models_i e_1\ e_2 \Longrightarrow v}$$

$$\textbf{(i-app2)}\ \frac{\begin{array}{c}\mathcal{E} \models_i e_2 \Longrightarrow v_2 \quad \mathcal{E} \models_i e_1 \Longrightarrow clos(\mathcal{E}', \mathtt{f}, \mathtt{x}, e) \\ \mathcal{E}'\{\mathtt{f} = clos(\mathcal{E}', \mathtt{f}, \mathtt{x}, e),\ \mathtt{x} = v_2\} \models_i e \Longrightarrow v\end{array}}{\mathcal{E} \models_i e_1\ e_2 \Longrightarrow v}$$

For constants and generic primitives we use semantic functions $cnst'$ and gen' respectively. Both $cnst'(c_\sigma)$ and $gen'(\mathtt{g}, \tau)$ return the compatible semantic value with the original functions, $cnst(c_\sigma)$ and $gen(\mathtt{g}, \tau)$. They are defined using the following translation function *strip* from GML semantic values for primitives P defined in Section 2.4.1 to semantic values of IML:

$$\begin{array}{l} strip(b_\sigma) = b_\sigma \\ strip(v_1, v_2) = strip(v_1), strip(v_2) \\ strip(clos(\emptyset, y, \mathtt{x}, e_t)) = clos(\emptyset, y, \mathtt{x}, strip(e_t)) \end{array}$$

Using *strip*, definitions of $cnst'$ and gen' are given as follows:

$$\begin{array}{l} cnst'(c_\sigma) = strip(cnst(c_\sigma)) \\ gen'(\mathtt{g}, \tau) = strip(gen(\mathtt{g}, \tau)) \end{array}$$

If there is no applicative rule in the above for an IML expression e under some evaluation environment $\mathcal{E}$, its evaluation returns an error, using the following (i-err):

$$\textbf{(i-err)}\ \frac{\text{When none of the above rules is applicative.}}{\mathcal{E} \models_i e \Longrightarrow err}$$

This is the only one rule for error. Our compilation algorithm will translate a well-typed fully type-annotated GML expression which never returns an error into an IML expression so that the both can evaluate and return compatible semantic values. This compilation correctness will be proven, therefore we do not need to define an independent error handling rule corresponds with each legal rule intensively, like what we have done for the fully type-annotated GML.

3.4 Compilation algorithm

3.4.1 Compilation of types

In execution of fully type-annotated GML expressions, when a derived generic instance is evaluated and its definition is expanded by (e-gvar) rule, the dynamic type variable annotations which are generalized in the definition are instantiated, so that the internal generic values can be evaluated under appropriate evaluation type contexts. In IML, such generalized dynamic type variables are replaced by identifiers and in evaluation they are substituted by run time types to emulate the instantiations in the GML semantics.

For each generalized dynamic type variable δ_i will be replaced by its special identifier, unique to each dynamic type variable. Because the set of identifiers is infinite, we can prepare such a special identifier for each δ_i which is distinct from the other special identifiers and the normal identifiers used in GML expressions. If we write a dynamic type variable δ_i, we presume that the identifier $\mathtt{d}_i$ is the special identifier for δ_i, and vice versa.

Before we define the compilation algorithm of GML expression to IML, we specify the algorithm for *type compilation* $\lfloor\tau\rfloor_V$, which translates type expressions to their corresponding IML expressions. This is not exactly same as «τ», because we need to replace the dynamic type variables δ_i to their corresponding identifiers $\mathtt{d}_i$ when they are in V, a set of generalized type variables. In the case they are not in V, they are replaced to «δ_i»:

$$
\begin{array}{rcl}
\lfloor base\rfloor_V & = & \text{«}base\text{»} \\
\lfloor\delta_i\rfloor_V & = & \mathtt{d}_i \quad \text{if } \delta_i \in V \\
 & & \text{Otherwise, «}\delta_i\text{»} \\
\lfloor\tau_1 \rightarrow \tau_2\rfloor_V & = & \mathtt{RTarrow(}\lfloor\tau_1\rfloor_V\mathtt{,}\lfloor\tau_2\rfloor_V\mathtt{)} \\
\lfloor\tau_1 * \tau_2\rfloor_V & = & \mathtt{RTtuple(}\lfloor\tau_1\rfloor_V\mathtt{,}\lfloor\tau_2\rfloor_V\mathtt{)}
\end{array}
$$

Type compilation for a regular type variable, $\lfloor\alpha\rfloor_V$ is not defined, but it does not matter since the type compilation algorithm $\lfloor\tau\rfloor_V$ is used against the "dynamized" type expressions which can be written $S(\delta)$ for some admissible substitution S and dynamic type variable δ in the compilation algorithm. Consequently $S(\delta)$ contains no regular type variable.

Run time type representation of type expressions are special cases of this type compilation. The run time type representation of a type expression τ is written as follows:

$$\text{«}\tau\text{»} = \lfloor\tau\rfloor_\emptyset$$

3.4.2 Compilation of expressions

The compilation algorithm $\lfloor e_t \rfloor_V$ takes a full annotated GML expression and a compilation environment, and returns its compilation as an IML expression:

$$
\begin{array}{rcl}
\lfloor c_\sigma : \tau \rfloor_V & = & c_\sigma \\
\lfloor \texttt{fun x} \rightarrow \ e_t \rfloor_V & = & \texttt{fun x} \ \rightarrow \lfloor e_t \rfloor_V \\
\lfloor e_{t1} \ e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V \ \lfloor e_{t2} \rfloor_V \\
\lfloor e_{t1}, e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V, \lfloor e_{t2} \rfloor_V \\
\lfloor \texttt{g}_\sigma : \tau \rfloor_V & = & \text{let } \sigma = \forall \delta_1 \ldots \delta_n . \tau' \\
& & \text{where } \tau \leq_S \sigma \text{ and } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
& & \text{in } \texttt{g} \ \lfloor S(\delta_1) \rfloor_V \ \ldots \ \lfloor S(\delta_n) \rfloor_V \\
\lfloor \texttt{x}_\sigma : \tau \rfloor_V & = & \text{let } \sigma = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n . \tau' \\
& & \text{where } \tau \leq_S \sigma \text{ and } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
& & \text{in } \texttt{x} \ \lfloor S(\delta_1) \rfloor_V \ \ldots \ \lfloor S(\delta_n) \rfloor_V \\
\multicolumn{3}{l}{\lfloor \texttt{let (rec) x}_{V'} \texttt{=} (e_{t1} : \tau) \ \texttt{in} \ e_{t2} \rfloor_V =} \\
& & \text{let } \forall V'.\tau = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n . \tau' \\
& & \text{where } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
& & \texttt{let (rec) x = fun d}_1 \ \ldots \ \texttt{d}_n \rightarrow \lfloor e_{t1} \rfloor_{V+V'} \ \texttt{in} \ \lfloor e_{t2} \rfloor_V
\end{array}
$$

As we explained in Section 3.2, generic primitives $\texttt{g}_\sigma : \tau$ and instances of generic values $\texttt{x}_\sigma : \tau$ are compiled to a function application of themselves to run time types for sending instantiation information. Derived generic let definitions are compiled with the abstractions which receive these run time types and their definition expressions must be compiled under the updated generalized type variable set $V + V'$. We can assume that V and V' are distinct by applying appropriate α-conversions of the generalized variable set if necessary. For the other constructs, the compilation just removes all the type annotations from expressions.

Example

Suppose that we have a generic primitive $\texttt{plus}_{\forall \delta . \delta \rightarrow \delta \rightarrow \delta}$. Then the following expression,

```
let double = fun x → plus x x
in double 1.0;;
```

has a typing e_t under the empty typing environment like:

```
let double{δ} = fun x → (plus:δ → δ → δ) x x
in (double∀δ.δ → δ:float → float) 1.0;;
```

and the compilation $\lfloor e_t \rfloor_\emptyset$ is as follows:

```
let double = fun d₁ x → plus d₁ x x
in double «float» 1.0;;
```

We extract some of its evaluation below (the entire evaluation requires too much space). The run time type «float» given to double will be sent to the generic primitive plus, and the expression plus $\mathtt{d}_1$ will be evaluated by (i-gen) to $gen'(\mathtt{plus}, \mathtt{float} \to \mathtt{float} \to \mathtt{float})$:

$$\begin{array}{rcl} \{\mathtt{d}_1 = \text{«}\mathtt{float}\text{»}\} \models_i \mathtt{plus}\ \mathtt{d}_1 & \Longrightarrow & gen'(\mathtt{plus}, \mathtt{float} \to \mathtt{float} \to \mathtt{float}) \\ & = & clos(\emptyset, \_, \mathtt{y}_1, \mathtt{fun}\ \mathtt{y}_2 \to \mathtt{y}_1\mathtt{+.}\mathtt{y}_2) \end{array}$$

gen' returns its explicitly defined closure for the plus function for floats, here it is equivalent to one for addition of floats (+.).

Using this closure the entire expression is evaluated to the result of plus $\mathtt{d}_1$ x x, 2.0.

$$\cfrac{\cfrac{\{\mathtt{x} = 1.0, \mathtt{y}_1 = 1.0\} \models_i (\mathtt{fun}\ \mathtt{y}_2 \to \mathtt{y}_1\mathtt{+.}\mathtt{y}_2)\ \mathtt{x} \Longrightarrow 2.0}{\{\mathtt{d}_1 = \text{«}\mathtt{float}\text{»}, \mathtt{x} = 1.0\} \models_i \mathtt{plus}\ \mathtt{d}_1\ \mathtt{x}\ \mathtt{x} \Longrightarrow 2.0}}{\cfrac{\cdots}{\{\mathtt{double} = clos(\ldots)\} \models_i \mathtt{double}\ \text{«}\mathtt{float}\text{»}\ 1.0 \Longrightarrow 2.0}}$$

□

3.5 Compilation correctness

What finally we argue in this chapter is the correctness of our compilation algorithm. Compiled code in IML must be evaluated to a "compatible" semantic value as the source GML code returns.

3.5.1 Semantic compilation compatibility

To extend the statement in the last paragraph to each step of evaluation, we need to define a relationship called *semantic compatibility* between semantic values for GML and IML. If an IML semantic value v' is semantically compatible to one of GML v, we write $v' \sqsubset v$. The definition of the relationship $\sqsubset$ for those semantic values is as follows in Figure 3.1. This relationship is also naturally extended to a relationship between evaluation environments of both systems. Here, the notation $\mathcal{E}\{S\}$ is for the evaluation environment $\mathcal{E}$ overridden by the evaluation bindings corresponds with S which is obtained by replacing its bindings for the dynamic type variables $(\delta_i \Rightarrow \tau_i)$ to $(\mathtt{d}_i \Rightarrow \text{«}\tau_i\text{»})$. The compatibility rule of normal and generic closures are rather complicated. These rules are intuitively read as follows:

Rule for normal closures There is a subexpression $\mathtt{fun}\ \mathtt{x} \to e_t$ in a program and typed as $\Gamma' \vdash \mathtt{fun}\ \mathtt{x} \to e_t : \tau_1' \to \tau_2'$. In evaluation, e_t is substituted to $S(e_t)$ and the function is evaluated to a closure $clos(E, y, \mathtt{x}, S(e_t))$. Then, the compilation of $\mathtt{fun}\ \mathtt{x} \to e_t$, $\mathtt{fun}\ \mathtt{x} \to \lfloor e_t \rfloor_S$ will be evaluated to some closure $clos(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S)$.

$$b_\sigma \sqsubset b_\sigma$$

$$\frac{v'_1 \sqsubset v_1 \quad v'_2 \sqsubset v_2}{v'_1, v'_2 \sqsubset v_1, v_2}$$

$$\frac{\begin{array}{c}\mathcal{E} \sqsubset E \quad \mathit{clos}(E, y, \mathtt{x}, S(e_t)) \in \tau_1 \to \tau_2 \\ \exists \Gamma, \quad E \in S(\Gamma) \quad \Gamma \vdash (\mathtt{fun\ x} \to e_t) : \tau'_1 \to \tau'_2 \ \text{ where } \ S(\tau'_1 \to \tau'_2) = \tau_1 \to \tau_2\end{array}}{\mathit{clos}(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S) \sqsubset \mathit{clos}(E, y, \mathtt{x}, S(e_t))}$$

$$\frac{\begin{array}{c}\mathcal{E} \sqsubset E \quad [\delta_i] = \mathit{Vlist}(V \cup \mathbb{D}, \tau) \\ \exists \Gamma, \quad \Gamma \vdash e_t \quad \forall V.\tau = \mathit{Gen}(\Gamma, e_t) \quad E \in S(\Gamma) \quad V \text{ is out of } S\end{array}}{\mathit{clos}(\mathcal{E}\{S\}, y, \mathtt{d}_1, \mathtt{fun\ d}_2 \ \ldots \ \mathtt{d}_n \to \lfloor e_t \rfloor_{\mathit{Dom}(S)+V}) \sqsubset \mathit{gclos}(E, y, V, S(e_t : \tau))}$$

$$\frac{\mathit{Dom}(E) = \mathit{Dom}(\mathcal{E}), \quad \forall \mathtt{x} \in \mathit{Dom}(E), \quad \mathcal{E}(\mathtt{x}) \sqsubset E(\mathtt{x})}{\mathcal{E} \sqsubset E}$$

$$\mathcal{E}\{S\} \equiv \mathcal{E}\{\mathtt{d}_i {\Rightarrow} «\tau_i» \mid (\delta_i {\Rightarrow} \tau_i) \in S\}$$

Figure 3.1: Semantic compilation compatibility

Rule for generic closures In a program, there is an expression e_t which define a derived generic value like `let g =` e_t `in ...` with a typing $\Gamma \vdash e_t$. In evaluation of the program, e_t is instantiated to $S(e_t)$ and stored in the generic closure $\mathit{gclos}(E, y, V, S(e_t))$. Then the compilation of the corresponding definition will be evaluated to some compatible closure $\mathit{clos}(\mathcal{E}\{S\}, y, \mathtt{d}_1, \mathtt{fun\ d}_2 \ \ldots \ \mathtt{d}_n \to \lfloor e_t \rfloor_{\mathit{Dom}(S)+V})$.

The semantic compatibility does not provide a one-to-one translation from a GML closure to one of IML. For non functional and non generic values, if $v' \sqsubset v$ then $v' = v$, but usually, to one GML closure $\mathit{clos}(E, y, \mathtt{x}, S(e_t))$ for example, we have more than one compatible IML closures $\mathit{clos}(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S)$ under the variety of the instantiations S. For example, look at the following two GML programs A and B:

A: `let double`$_{\{\delta\}}$ `= fun x` $\to$ `plus x x in double 1;;`

B: `let double`$_\emptyset$ `= fun x` $\to$ `(plus : int` $\to$ `int` $\to$ `int) x x in double 1;;`

Both define functions named `double` using `plus` generic primitive. The program A defines a derived generic function `double` whose type is $\forall \delta.\delta \to \delta$, while B instantiates `plus` to `int` $\to$ `int` $\to$ `int` in the definition, and therefore `double` is defined as a non generic function whose type is `int` $\to$ `int`. In the both program, the use of `double` in the expression `double 1` is evaluated to the same closure $c = \mathit{clos}(E, \_, \mathtt{x}, (\mathtt{plus} : \mathtt{int} \to \mathtt{int} \to \mathtt{int})\ \mathtt{x}\ \mathtt{x})$:

$$\text{A:} \quad \frac{E \models \{\delta {\Rightarrow} \mathtt{int}\}(\mathtt{fun\ x} \to (\mathtt{plus} : \delta \to \delta \to \delta)\ \mathtt{x}\ \mathtt{x}) \Longrightarrow c}{E\{\mathtt{double} = \mathit{gclos}(E, \_, \{\delta\}, \mathtt{fun\ x} \to \mathtt{plus\ x\ x})\} \models \mathtt{double} : \mathtt{int} \to \mathtt{int} \Longrightarrow c}$$

$$\text{B:} \quad \frac{E \models \mathtt{fun\ x} \to (\mathtt{plus} : \mathtt{int} \to \mathtt{int} \to \mathtt{int})\ \mathtt{x}\ \mathtt{x} \Longrightarrow c}{E\{\mathtt{double} = c\} \models \mathtt{double} : \mathtt{int} \to \mathtt{int} \Longrightarrow c}$$

Though their results are the same in the GML semantics, in the evaluations of their compilations:

Compilation of A and its evaluation of `double «int»`:

```
let double = fun ty x → plus ty x x in double «int» 1;;
```

$$\frac{\mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{double} \Longrightarrow c_{\mathrm{A}} \quad \text{where } c_{\mathrm{A}} = \mathit{clos}(\mathcal{E}, \_, \texttt{ty}, \texttt{fun x} \rightarrow \texttt{plus ty x x}) \qquad \mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{«int»} \Longrightarrow \texttt{«int»}}{\mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{double «int»} \Longrightarrow \mathit{clos}(\mathcal{E}\{\texttt{ty} = \texttt{«int»}\}, \_, \texttt{x}, \texttt{plus ty x x})}$$

Compilation of B and its evaluation of `double`:

```
let double = fun x → plus «int» x x in double 1;;
```

$$\frac{\mathcal{E} \models_i \texttt{fun x} \rightarrow \texttt{plus «int» x x} \Longrightarrow c_{\mathrm{B}} \quad \text{where } c_{\mathrm{B}} = \mathit{clos}(\mathcal{E}, \_, \texttt{x}, \texttt{plus «int» x x})}{\mathcal{E}\{\texttt{double} = c_{\mathrm{B}}\} \models_i \texttt{double} \Longrightarrow c_{\mathrm{B}}}$$

The resulting closures of these two compilations which are applied to `1` are slightly different:

$$\begin{array}{lllll} \text{A:} & \mathit{clos}(\mathcal{E}\{\texttt{ty} = \texttt{«int»}\}, & \_, \texttt{x}, & \texttt{plus ty} & \texttt{x x}) \\ \text{B:} & \mathit{clos}(\mathcal{E}, & \_, \texttt{x}, & \texttt{plus «int»} & \texttt{x x}) \end{array}$$

But apparently the closure of B is a partially evaluated form of A and they work same as functions. Both closures are considered semantically compatible with the GML closure

$$\mathit{clos}(E, \_, \texttt{x}, (\texttt{plus} : \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int})\ \texttt{x x})$$

Like semantic values we can also have one or more semantically compatible evaluation environments $\mathcal{E}$ to one E.

3.5.2 Compilation correctness

The correctness of the compilation algorithm is now proven using the semantic compatibility:

Theorem 11 (Compilation correctness) *If a fully type-annotated expression e_t has a typing $\emptyset \vdash e_t$ and $\emptyset \models e_t \Longrightarrow v$, then its compilation in IML under the empty compilation environment $\lfloor e_t \rfloor_\emptyset$ is evaluated to v' which is compatible with v without error, i.e.*

$$\emptyset \models_i \lfloor e_t \rfloor_\emptyset \Longrightarrow v' \sqsubset v$$

Like Theorem 4, to prove this theorem we need to enforce the statements for each evaluation step of GML expressions:

Theorem 12 (Stronger compilation correctness) *Let e_t be a fully type-annotated GML expression which has a typing under a typing environment Γ, i.e, $\Gamma \vdash e_t$. Let S be a substitution and E be a GML evaluation environment such that $E \in S(\Gamma)$. If $E \models S(e_t) \Longrightarrow v$ for some v then for any IML evaluation environment $\mathcal{E}$ such that $\mathcal{E} \sqsubset E$,*

$$\mathcal{E}\{S\} \models_i \lfloor e_t \rfloor_S \Longrightarrow v'$$

such that $v' \sqsubset v$.

Proof: First we prove that $E \models S(e_t)$ always succeeds. From Theorem 3, $S(\Gamma) \vdash S(e_t)$. Since $E \in S(\Gamma)$, from Theorem 5, the evaluation $S(e_t)$ under E always succeeds and returns a semantic value v: $E \models S(e_t) \Longrightarrow v$.

The main proof is done by induction over the evaluation derivation $E \models S(e_t) \Longrightarrow v$.

(e-cnst) e_t must be a constant: let $e_t = c_\sigma : \tau$.

Typing and evaluation in GML:
Its typing is by (t-cnst):

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

Because σ is closed, $S(e_t) = c_\sigma : S(\tau)$. The typing of $S(e_t)$ becomes as follows, for some substitution S_1:

$$\frac{S(\tau) \leq_{S_1} \sigma}{S(\Gamma) \vdash c_\sigma : S(\tau)}$$

The constant is evaluated by (e-cnst):

$$E \models c_\sigma : S(\tau) \Longrightarrow S_1(cnst(c_\sigma))$$

Evaluation of the compiled code:
The compilation of e_t is:

$$\lfloor e_t \rfloor_S = \lfloor c_\sigma : \tau \rfloor_S = c_\sigma$$

and it is evaluated in the IML evaluation as follows, which is compatible to $S_1(cnst(c_\sigma))$ by the definition of $cnst'$: $cnst'(c_\sigma) = strip(cnst(c_\sigma))$.

$$\mathcal{E}\{S\} \models_i c_\sigma \Longrightarrow cnst'(c_\sigma) \sqsubset S_1(cnst(c_\sigma))$$

(e-gen) From the used evaluation rule, e_t must be an instance of a generic primitive: $e_t = \mathsf{g}_\sigma : \tau$.

Typing and evaluation in GML:

The original typing $e_t = \mathsf{g}_\sigma : \tau$ is as follows:

$$\frac{\tau \leq_{S_0} \sigma}{\Gamma \vdash \mathsf{g}_\sigma : \tau}$$

and if we let $\forall V.\tau' = \sigma$, the typing and the evaluation of $S(e_t)$ are:

$$\frac{S_1 = (S \circ S_0)_{|V} \quad S(\tau) \leq_{S_1} \sigma}{S(\Gamma) \vdash \mathsf{g}_\sigma : S(\tau)}$$

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

Evaluation of the compiled code:

If we let $[\delta_i] = Vlist(V, \tau')$, the expression e_t is compiled to:

$$\lfloor e_t \rfloor_S = \lfloor \mathsf{g}_\sigma : \tau \rfloor_S = \mathsf{g}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S$$

Each argument $\lfloor S_0(\delta_i) \rfloor_S$ is evaluated by (i-runtype) as follows:

$$\mathcal{E}\{S\} \models_i \lfloor S_0(\delta_i) \rfloor_S \Longrightarrow «(S \circ S_0)(\delta_i)»$$

It is because $\lfloor \tau \rfloor_S$ replaces the occurrences of $\delta_i \in Dom(S)$ to its special variable d_i, and $\mathcal{E}\{S\}$ has a binding $(\mathsf{d}_i \Rightarrow «S(\delta_i)»)$ from its definition. Therefore the entire compilation is evaluated by (i-gen) as follows:

$$\mathcal{E}\{S\} \models_i \mathsf{g}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow gen'(\mathsf{g}, \{\delta_i \Rightarrow (S \circ S_0)(\delta_i)\}(\tau'))$$

Since $\tau \leq_{S_0} \sigma = \forall V.\tau'$ and $\{\delta_i\} = V$,

$$gen'(\mathsf{g}, \{\delta_i \Rightarrow (S \circ S_0)(\delta_i)\}(\tau')) = gen'(\mathsf{g}, S(\tau))$$

$gen'(\mathsf{g}, S(\tau)) \sqsubset gen(\mathsf{g}, S(\tau))$ from the definition of gen'.

(e-nvar) Let $e_t = \mathtt{x}_\sigma : \tau$.

Typing and evaluation in GML:

Suppose we have the typing for $e_t = \mathtt{x}_\sigma : \tau$ by (t-inst) for some substitution S_0:

$$\frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq_{S_0} \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau}$$

From Theorem 3 we have the following typing of $S(e_t)$:

$$\frac{(S(\Gamma))(\mathtt{x}) = S(\sigma) \quad S(\tau) \leq_{S_1} S(\sigma)}{S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)}$$

where $S_1 = (S \circ S_0)_{|BV(\sigma)}$. Because our rule used here is (e-nvar), $E(\mathtt{x})$ must be a regular value v, not a generic closure. Since $S(\tau) \leq_{S_1} S(\sigma)$,

$$\frac{E(\mathtt{x}) = v}{E \models \mathtt{x}_{S(\sigma)} : S(\tau) \Longrightarrow S_1(v)}$$

Evaluation of the compiled code:

Because $BDV(\sigma) = \emptyset$, the compilation of e_t is simple:

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_S = \mathtt{x}$$

Since $\mathcal{E} \sqsubset E$, $\mathcal{E}(\mathtt{x}) = v' \sqsubset v$. Hence,

$$\mathcal{E}\{S\} \models_i \mathtt{x} \Longrightarrow v' \sqsubset S(v)$$

(e-gvar) Like in the case of (e-nvar) $e_t = \mathtt{x}_\sigma : \tau$, but its binding in the evaluation environment is a generic closure.

Typing and evaluation in GML:

The typings of $e_t = \mathtt{x}_\sigma : \tau$ and $S(e_t)$ are as follows:

$$\frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq_{S_0} \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau}$$

$$\frac{(S(\Gamma))(\mathtt{x}) = S(\sigma) \quad V = BV(\sigma) \quad S_1 = (S \circ S_0)_{|V} \quad S(\tau) \leq_{S_1} S(\sigma)}{S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)}$$

Because we use (e-gvar), $BDV(\sigma) = V \cap \mathbb{D}$ must not empty and $E(\mathtt{x})$ is bound by a generic closure which can be written as $gclos(E', \_, V, e'_t)$:

$$\frac{E(\mathtt{x}) = gclos(E', \_, V, e'_t) \quad E' \models S_1(e'_t) \Longrightarrow v}{E \models \mathtt{x}_{S(\sigma)} : S(\tau) \Longrightarrow v}$$

Induction step:

Because $\mathcal{E} \sqsubset E$, $\mathcal{E}(\mathtt{x}) \sqsubset E(\mathtt{x}) = gclos(E', \_, V, e'_t)$. From the definition of the semantic compilation compatibility,

$$\mathcal{E}(\mathtt{x}) = clos(\mathcal{E}'\{S'\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2\ \ldots\ \mathtt{d}_n \to \lfloor e''_t \rfloor_{Dom(S')+V})$$

where

$$\begin{array}{l} S'(e''_t) = e'_t \\ E(\mathtt{x}) = gclos(E', \_, V, e'_t) = gclos(E', \_, V, S'(e''_t)) \\ \exists \Gamma',\ \Gamma' \vdash e''_t : \tau'' \ \ \forall V.\tau'' = Gen(\Gamma', e''_t) \\ E' \in S'(\Gamma'), \quad V \text{ is out of } S' \end{array}$$

Since $Dom(S_1) = V$ and S' is out of V, $S_1 \circ S' = S_1 + S'$. Now the evaluation of $S_1(e'_t)$ is rewritten as follows:

$$E' \models (S_1 + S')(e''_t) \Longrightarrow v$$

From the generalization of e''_t, $FV(\Gamma') \cap V = \emptyset$. Therefore, from $Dom(S_1) = V$, $(S_1 + S')(\Gamma) = S'(\Gamma)$. Because $E' \in S'(\Gamma')$,

$$E' \in (S_1 + S')(\Gamma') = S'(\Gamma)$$

From this and the typing of $\Gamma' \vdash e''_t$, we can use the induction hypothesis for the evaluation of e''_t:

$$\mathcal{E}'\{S_1 + S'\} \models_i \lfloor e''_t \rfloor_{Dom(S')+V} \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:
The expression is compiled into the following IML expression:

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_S = \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S$$

Each argument $\lfloor S_0(\delta_i) \rfloor_S$ is evaluated as follows, by the same argument we did in (e-gen) part:

$$\mathcal{E}\{S\} \models_i \lfloor S_0(\delta_i) \rfloor_S \Longrightarrow «(S \circ S_0)(\delta_i)»$$

Since $\mathcal{E}'\{S_1 + S'\} = \mathcal{E}'\{S'\}\{\mathsf{d}_i = «S_1»(\delta_i)\} = \mathcal{E}'\{S'\}\{\mathsf{d}_i = «(S \circ S_0)»(\delta_i)\}$, using (i-app) and (i-fun) n times, we obtain the evaluation of the compilation from the result of the induction:

$$\cfrac{\cfrac{\mathcal{E}'\{S'\}\{\mathsf{d}_i = «(S \circ S_0)(\delta_i)»\} \models_i \lfloor e''_t \rfloor_{S_1+S'} \Longrightarrow v'}{\ldots}}{\mathcal{E}\{S\} \models_i \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-tpl) Let $e_t = e_{t1}\ e_{t2}$.

Typing and evaluation in GML:
We have the following typing of $S(e_t) = S(e_{t1}, e_{t2})$:

$$\frac{S(\Gamma) \vdash S(e_{t1}) \qquad S(\Gamma) \vdash S(e_{t2})}{S(\Gamma) \vdash S(e_{t1}, e_{t2})}$$

and its evaluation:

$$\frac{E \models S(e_{t1}) \Longrightarrow v_1 \qquad E \models S(e_{t2}) \Longrightarrow v_2}{E \models S(e_{t1}, e_{t2}) \Longrightarrow v_1, v_2}$$

Induction step:
By applying the induction hypothesis to both premises, we get:

$$\mathcal{E}'\{S\} \models_i \lfloor e_{tj} \rfloor_S \Longrightarrow v'_j \sqsubset v_j \quad \text{for } j = 1, 2$$

Evaluation of the compiled code:
The expression e_{t1}, e_{t2} is compiled as follows:

$$\lfloor e_{t1}, e_{t2} \rfloor_S = \lfloor e_{t1} \rfloor_S, \lfloor e_{t2} \rfloor_S$$

and evaluated in IML system using (i-tpl):

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v'_1 \qquad \mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2}{\mathcal{E}\{S\} \models_i \lfloor e_{t1}, e_{t2} \rfloor_S \Longrightarrow v'_1, v'_2}$$

Clearly, $(v_1', v_2') \sqsubset (v_1, v_2)$.

(e-fun) Let $e_t = \texttt{fun x} \rightarrow e_t'$.

Typing and evaluation in GML:
The typing and evaluation of $S(e_t) = \texttt{fun x} \rightarrow S(e_t')$ are:

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

$$E \models S(\texttt{fun x} \rightarrow e_t') \Longrightarrow \mathit{clos}(E, \_, \texttt{x}, S(e_t'))$$

where $\mathit{clos}(E, \_, \texttt{x}, S(e_t')) \in S(\tau_1 \rightarrow \tau_2)$ from the Theorem 5.

Evaluation of the compiled code:
The compilation of the expression $\texttt{fun x} \rightarrow e_t'$ is:

$$\lfloor \texttt{fun x} \rightarrow e_t' \rfloor_S = \texttt{fun x} \rightarrow \lfloor e_t' \rfloor_S$$

and it is evaluated as follows:

$$\mathcal{E}\{S\} \models_i \texttt{fun x} \rightarrow \lfloor e_t' \rfloor_S \Longrightarrow \mathit{clos}(\mathcal{E}\{S\}, \_, \texttt{x}, \lfloor e_t' \rfloor_S)$$

Since $\Gamma \vdash \texttt{fun x} \rightarrow e_t' : \tau_1 \rightarrow \tau_2$, from the definition of the semantic compilation compatibility between closures,

$$\mathit{clos}(\mathcal{E}\{S\}, \_, \texttt{x}, \lfloor e_t' \rfloor_S) \sqsubset \mathit{clos}(E, \_, \texttt{x}, S(e_t'))$$

(e-let) This time $e_t = \texttt{let x}_V \texttt{=} e_{t1} \texttt{ in } e_{t2} : \tau_2$. Since it is evaluated by (e-let), the generalized variable set V must not contain dynamic type variables.

Typing and evaluation in GML:
We have the typing of the expression e_t:

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

where $V \cap \mathbb{D} = \emptyset$. Then one for $S(e_t)$ is as follows (we use the renaming of V so that V is out of S like we did in the proof of Theorem 3):

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

It is evaluated as below:

$$\frac{E \models S(e_{t1}) \Longrightarrow v_1 \quad E\{\texttt{x} = v_1\} \models S(e_{t2}) \Longrightarrow v_2}{E \models \texttt{let x}_V \texttt{=} S(e_{t1}) \texttt{ in } S(e_{t2}) \Longrightarrow v_2}$$

Induction step:

By using the induction hypothesis for the evaluation of $S(e_{t1})$, the compilation is evaluated to some v_1' which is compatible to v_1:

$$\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v_1' \sqsubset v_1$$

From the semantic typing of the evaluation $E \models S(e_{t1} : \tau_1) \Longrightarrow v_1$, $v_1 \in S(\tau_1)$. From Lemma 9, $v_1 \in \forall V.S(\tau_1) = S(\forall V.\tau_1)$. Since $E \in S(\Gamma)$,

$$E\{\mathtt{x} = v_1\} \in S(\Gamma)\{\mathtt{x} : S(\forall V.\tau_1)\} = S(\Gamma\{\mathtt{x} : \forall V.\tau_1\})$$

From this and the typing of

$\Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2}$ and the evaluation $E\{\mathtt{x} = v_1\} \models S(e_{t2}) \Longrightarrow v_2$, we can use the induction hypothesis and obtain:

$$\mathcal{E}\{\mathtt{x} = v_1'\}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v_2' \sqsubset v_2$$

Evaluation of the compiled code:

The compilation of the entire expression is as follows:

$$\lfloor \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} \rfloor_S = \mathtt{let}\ \mathtt{x}_V = \lfloor e_{t1} \rfloor_S\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S$$

and evaluated by (i-let), using the results of the inductions.

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v_1' \quad \mathcal{E}\{S\}\{\mathtt{x} = v_1'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v_2'}{\mathcal{E}\{S\} \models_i \mathtt{let}\ \mathtt{x}_V = \lfloor e_{t1} \rfloor_S\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S \Longrightarrow v_2' \sqsubset v_2}$$

(e-g-let) The expression takes the same form as the case of (e-let), $e_t = \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2$. But in this case V contains dynamic type variables.

Typing and evaluation in GML:

We have the same typing as the case of (e-let). The difference is that $V \cap \mathbb{D} \neq \emptyset$. $S(e_t)$ is now evaluated differently, by (e-g-let):

$$\frac{E\{\mathtt{x} = gclos(E, \_, V, S(e_{t1}))\} \models S(e_{t2}) \Longrightarrow v}{E \models \mathtt{let}\ \mathtt{x}_V = S(e_{t1})\ \mathtt{in}\ S(e_{t2}) \Longrightarrow v}$$

Induction step:

From the typing of e_{t1}, we have

$$\Gamma\{\mathtt{x} = \forall V.\tau_1\} \vdash e_{t1}$$

Let

$$c = gclos(E, \_, V, S(e_{t1}))$$
$$c' = clos(\mathcal{E}\{S\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2\ \ldots\ \mathtt{d}_n \rightarrow \lfloor e_t \rfloor_{Dom(S)+V})$$

Then from the typing of $S(e_{t1})$, using (s-gclos), $c \in S(\forall V.\tau_1)$. Therefore,

$$E\{\mathtt{x} = c\} \in S(\Gamma\{\mathtt{x} = \forall V.\tau_1\})$$

From the definition of the semantic compilation compatibility, we have $c' \sqsubset c$, and this leads to:

$$\mathcal{E}\{\mathtt{x} = c'\} \sqsubset E\{\mathtt{x} = c\}$$

From these we can use the induction hypothesis for the evaluation $E\{\mathtt{x} = c\} \models S(e_{t2}) \Longrightarrow v$ and obtain

$$\mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:

The compilation of e_t becomes:

$$\lfloor \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} \rfloor_S = \mathtt{let}\ \mathtt{x}_V = \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_n \to \lfloor e_{t1} \rfloor_{Dom(S)+V}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S$$

and its evaluation is as follows by (i-let) using the result of the induction:

$$\frac{\mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'}{\mathcal{E}\{S\} \models_i \mathtt{let}\ \mathtt{x}_V = \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_n \to \lfloor e_{t1} \rfloor_{Dom(S)+V}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-app1) Let $e_t = e_{t1}\ e_{t2}$.

Typing and evaluation in GML:

The typing of $S(e_t) = S(e_{t1})\ S(e_{t2})$ is as follows:

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

Its evaluation is done by (e-app1):

$$\frac{\begin{array}{c} E \models S(e_{t2}) \Longrightarrow v_2 \qquad E \models S(e_{t1}) \Longrightarrow clos(E', \_, \mathtt{x}, e'_t) \\ E'\{\mathtt{x} = v_2\} \models e'_t \Longrightarrow v \end{array}}{E \models S(e_{t1}\ e_{t2}) \Longrightarrow v}$$

Induction step:

Applying the induction hypothesis to the first evaluation premise, we get:

$$\mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2 \sqsubset v_2$$

We also use the induction hypothesis for the second evaluation premise

$$E \models S(e_{t1}) \Longrightarrow clos(E', \_, \mathtt{x}, e'_t)$$

there is a typing environment Γ', a substitution S' and an expression e''_t such that

$$\begin{array}{l} S'(e''_t) = e'_t \\ \exists\Gamma',\quad \Gamma' \vdash (\texttt{fun x} \to e''_t) : \tau'_1 \to \tau'_2 \text{ where } S'(\tau'_1 \to \tau'_2) = S(\tau_1 \to \tau_2) \\ E' \in S'(\Gamma') \\ \mathcal{E}' \sqsubset E' \end{array}$$

and

$$\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow clos(\mathcal{E}'\{S'\}, \_, \texttt{x}, \lfloor e''_t \rfloor_{S'}) \sqsubset clos(E', \_, \texttt{x}, S'(e''_t))$$

The typing of $\texttt{fun x} \to e''_t$ must be deduced by (t-fun):

$$\frac{\Gamma'\{\texttt{x} : \tau'_1\} \vdash e''_t : \tau'_2}{\Gamma \vdash \texttt{fun x} \to e''_t : \tau'_1 \to \tau'_2}$$

Since $S'(\tau'_1) = S(\tau_1)$, $v_2 \in S'(\tau'_1) = S(\tau_1)$. From $E' \in S'(\Gamma')$,

$$E'\{\texttt{x} = v_2\} \in S'(\Gamma'\{\texttt{x} : \tau'_1\})$$

From $\mathcal{E}' \sqsubset E'$ and $v'_2 \sqsubset v_2$, we have

$$\mathcal{E}'\{\texttt{x} = v'_2\} \sqsubset E'\{\texttt{x} = v_2\}$$

Now we can apply the induction hypothesis to the evaluation of $e'_t = S(e''_t)$ and obtain:

$$\mathcal{E}'\{S'\}\{\texttt{x} = v'_2\} \models \lfloor e''_t \rfloor_{S'} \Longrightarrow v'$$

where $v' \sqsubset v$

Evaluation of the compiled code:
The compilation of e_t is as follows:

$$\lfloor e_{t1}\ e_{t2} \rfloor_S = \lfloor e_{t1} \rfloor_S\ \lfloor e_{t2} \rfloor_S$$

and evaluated using the results and (i-app1):

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2 \quad \mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow clos(\mathcal{E}'\{S'\}, \_, \texttt{x}, \lfloor e'_t \rfloor_{S'}) \quad \mathcal{E}'\{S'\}\{\texttt{x} = v'_2\} \models_i \lfloor e'_t \rfloor_{S'} \Longrightarrow v'}{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S\ \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-gvar-r), (e-let-r), (e-g-let-r) and (e-app2) All are provable in the same way as their non-recursive versions.

$\square$

Now Theorem 11 is just a special case of Theorem 12, when $\Gamma = \emptyset, E = \emptyset, S = \emptyset$.

3.6 Coherence problem of generic primitives

As Ohori[Oho89] pointed out, in the analysis of the denotational semantics of implicitly typed ML language, if we build a semantic system over explicitly typed ML terms, different typing derivations of an implicitly typed program may provide incompatible semantics each other. This problem is know as the *coherence problem*[BTCGS94]. But in the actual execution of ML programs, this coherence problem is not a big trouble, since programs are evaluated under the semantics of the untyped lambda calculus. Ignoring all the type informations, behavior of a program at execution does not depend on its typing. However, as Jones discussed in an ML type system extension for qualified types[?], the coherence has greater importance once we introduce type-dependent semantics of polymorphic values, since different typings of polymorphic instances may provide completely different semantics at execution.

Our language has generic primitives with type-dependent semantics. Therefore, we cannot directly define semantics of a bare expression without type annotations. Its semantics is obtained through the semantics of the type-annotated GML, by mapping the bare expression to a fully type-annotated expression. If a bare expression has more than one possibility of typing, these typings may provide different semantics incompatible each other. For example, consider a generic primitive $\mathtt{g}_{\forall\delta.\delta\ \mathtt{list}}$ with the following semantics:

$$
\begin{array}{lll}
gen(\mathtt{g},\ \mathtt{int\ list}) & \leadsto & \mathtt{[1]} \\
gen(\mathtt{g},\ \mathtt{string\ list}) & \leadsto & \mathtt{["hello";\ "world"]} \\
\ldots & &
\end{array}
$$

Then a source program `List.length g` is *incoherent*, since depending on its typing derivations, it has different semantics and returns different results:

```
List.length (g : int list)    ⇝ 1
List.length (g : string list) ⇝ 2
```

Incoherence of programs must be detected by type checker, since we cannot fix their semantics.

The coherence of a program can be roughlydefined as follows:

Definition 7 (Coherence) *A bare program e is coherent, when it is typable: $W(\emptyset, e) = e_t$ and for any substitution S, $\lfloor e_t \rfloor_\emptyset \equiv \lfloor S(e_t) \rfloor_\emptyset$.*

Looking at the compilation algorithm $\lfloor \cdot \rfloor_V$, we can easily obtain the following theorem:

Theorem 13 (Free dynamic type variables cause incoherence) *A program e is coherent when its principal typing $e_t = W(\emptyset, e)$ contains no free dynamic type variable occurrence: $DV(e_t) = \emptyset$.*

In other words, when the algorithm W finds a principal typing of a given program e such that $e_t = W(\emptyset, e)$ and e_t contains free occurrences of dynamic type variables, e is incoherent. In the case of incoherence, the programmer must explicitly constrain types of incoherent generic instances to remove the incoherence and provide appropriate semantics to the program. The type inference may be able to choose one of possible typings by instantiating the free dynamic type variable occurrences to some closed types in order to provide a default semantics automatically of an incoherent program, but there is always a risk that such "default" can be incompatible with the programmer's intension. Therefore, the system must print out a warning message if it performs this auto-elimination of incoherence. Chapter 10 will discuss again about the detection of incoherence using *generic constraint typing system* (Chapter 8) which prevents incoherent instantiations of generic values.

Future work around coherence

The definition of coherence Definition 7 is not complete, though the following result Theorem 13 is correct. We considers only the typings obtained by substituting free type variables from the principal typing $e_t = W(\emptyset, e)$ here. For the precise discussion, the other possible typings must be also taken into account, introducing equational theory between compiled code. Also, the coherence can be discussed over the semantics of type-annotated GML defined in Section 2.4, instead of using compilation equality. These formalization is left as a future work.

3.7 Conclusion

We defined a compilation algorithm of GML to IML, an untyped ML like language. The operational semantics of GML with type annotations and instantiations are now translated to classical typeless semantics of IML which can be compiled to efficient executable programs by existing ML compilation techniques.

Type instantiation mechanism of the GML semantics system is translated to run time type dispatching in IML. Evaluation type contexts of generic values are passed as run time types, normal ML value representation of type expressions through functional applications. Generic values are compiled to functions which take these dispatched run time types to reconstruct their evaluation type context. This idea is exactly same as the explicit type abstractions and applications in System F [GTL90], where the definition of `double` whose type scheme is $\forall\delta.\delta \rightarrow \delta$ can be written as $\Lambda\delta.\lambda\mathtt{x}.\mathtt{plus}\ \delta\ \mathtt{x}\ \mathtt{x}$, for example. In the compilation of extensional polymorphism, however, the type abstractions of regular type variables $\Lambda\alpha$ and type applications to them are removed for efficiency, since polymorphism by regular type variables is parametric and its instantiation does not affect the semantics. Therefore, run time type dispatching is performed only where it is really required by generic values that are quantified by dynamic type variables. Non-generic part of programs is compiled to the identical expressions without type annotations, therefore they are evaluated as efficiently as in the original ML system, without any overhead.