Chapter 4

Dynamic values

4.1 Introduction

Dynamic values — an example of generic primitives

In the previous chapters, the semantics of the generic primitives are defined explicitly. We have defined their typing, but never discussed about the internals — semantics. In this chapter, we try to give an important example of $gen(\mathtt{g}, \tau)$ — the primitives for *dynamic values* [ACPR92, LM94]. Dynamic values provide a way to perform run-time typing on values. This mechanism is provided by two primitives, one for creating dynamic values and another for destructing them. These two primitives are often called as `dynamic` and `coerce` respectively. The expression `dynamic` v creates a dynamic value from v, that is a pair (v, τ) which consists of v and its type τ. This $(\text{value}, \text{type})$ pair is encapsulated into an abstract value of the abstract type ***`dyn`*** for the dynamic values, hiding the original type of the value. `coerce` expression destructs these dynamic values and can handle their contents differently depending on their types, performing simple run time type check. This type hiding capability is useful to simulate heterogeneous data structure and overloaded functions in ML, even the cost of manual encapsulation and decapsulation of dynamic values.

Dynamic primitives and polymorphic dynamization

We introduce the `dynamic` and `coerce` primitives as generic primitives and see how easily their basic functionality can be implemented in extensional polymorphism framework; the run time type dispatching in the compilation of extensional polymorphism provides these primitives almost for free. Then we try polymorphic dynamic encapsulation: encapsulation of polymorphic values in dynamic values that can be coerced back again to polymorphic values. Since this is not always possible, we must extend the typing for dynamic primitives so that they can perform generalization checks of the types of dynamized values.

Outline of this chapter

Section 4.2 introduces the primitives for dynamic values, `dynamic` and `coerce` as generic primitives, and give their semantics as the forms of compiled expression in the intermediate language IML. From Section 4.3 we add polymorphic encapsulation and decapsulation capability to these primitives. Section 4.4 explains required type system modifications for polymorphic dynamization. Then in Section 4.5, `coerce` primitive is extended to a special expression construct to decapsulate the contents of dynamized values to polymorphic identifiers.

4.2 Dynamic values in extensional polymorphism

4.2.1 Dynamic values

Dynamic values provide simple typing of values at run time. This mechanism is provided by two primitives, known as `dynamic` and `coerce` (or `typecase`). Usually `dynamic` primitive receives a value v and returns a dynamic value which is the pair (v, τ), v itself and its type τ. This pair is abstracted as a value of the abstract type `dyn`, so that we can hide the type information. To destruct the dynamic values and retrieve their content, `coerce` primitive checks whether their internal type information τ is "compatible" to the expected type context τ' (how the coerced content is used). Intuitively, if τ is more general than τ', τ can be considered as compatible to τ'. If incompatible, the `coerce` primitive rejects the coercion of the dynamic values. The following code shows that a dynamic encapsulation of integer. The dynamic values of 1 can be coerced back to integer, but not to float:

```
# let d = dynamic 1;; (* creates a pair (1,int) *)
d : dyn = <abstr>
# coerce d + 1;; (* coercion to int, compatible *)
- : int = 2
# (coerce d : float);; (* coercion to float, incompatible *)
Uncaught_exception: Coercion_failure (* An exception instead of Seg. fault *)
```

The coercion may be able to take multiple cases. In this case, `coerce` is no longer a primitive function, but a special expression syntax: The expression

$$
\begin{array}{l}
\texttt{coerce } d_1\texttt{,...,}d_n \texttt{ with} \\
\texttt{| (x}_{11} \texttt{ : } \tau_{11}\texttt{),..., (x}_{1n} \texttt{ : } \tau_{1n}\texttt{)} \rightarrow e_1 \\
\texttt{| ...} \\
\texttt{| (x}_{m1} \texttt{ : } \tau_{m1}\texttt{),..., (x}_{mn} \texttt{ : } \tau_{mn}\texttt{)} \rightarrow e_m
\end{array}
$$

first tries to coerce n dynamic values $d_1, \ldots, d_n$ to $\tau_1, \ldots, \tau_n$ respectively, and if their coercion are all possible, the contents of the dynamic values are bound to the pattern variables $\mathtt{x}_i$ to evaluate e_1. Otherwise, it tries the other cases. If none of the cases matches with the type information of $d_1, \ldots, d_n$, it raises a run time exception.

Even with the cost of the run time type checking at coercion, the dynamic values are sometimes useful to unify various typed values into one `dyn` type. For example, using dynamic values we can simulate heterogeneous lists as follows:

```
# let hlist = [dynamic 1; dynamic 3.14; dynamic "hello"];;
hlist : dyn list = [ <abstr>; <abstr>; <abstr> ]
```

The created list `hlist` has the type `dyn list` which is totally monogeneous, but the types of the elements are stored inside dynamic values. To retrieve elements from this heterogeneous list, we have to coerce manually, of course:

```
# let print_dyn d =
    coerce d with
    | x : int ⇒ print_int x
    | x : float ⇒ print_float x
    | x : string ⇒ print_string x
print_dyn : dyn → unit
# List.iter print_dyn hlist;;
1 3.14 hello
- : unit = ()
```

The above code simulates an overloaded function using dynamic values. This overloaded behavior is quite similar to generic values in extensional polymorphism. However, in the ability of defining overloaded values, dynamics clearly fall behind extensional polymorphism.

First, to use these dynamic generic functions we require explicit use of the `dynamic` primitive to give their arguments, like `print_dyn (dynamic 1)`. In extensional polymorphism framework, the run time type dispatch for generic values is automatic in the compilation, as we have seen in Chapter 3.

Second, hiding the types of encapsulated values makes it impossible to detect illegal applications of dynamic generic functions statically. For example, when we are given an expression `print_dyn d`, whether `d` has a valid type for `print_dyn` (i.e. `int`, `float` or `string`) is not known until at run time evaluation when we have `d` at hand. On the other hand, the user-defined generic values which will be available in Chapter 7 can also define non-total overloaded values, but their instance validity can be statically checked by the generic constraint typing system later introduced in Chapter 8.

In spite of these disadvantages compared to generics, dynamic values are still very interesting and important in extensional polymorphism. First of all, we can obtain its basic primitives almost for free in extensional polymorphism, as already proposed in [DRW95]. In addition, the abstract type `dyn` can provide a powerful type reconstruction tactics called *dyn-unification* to solve ambiguity problems of generic constraint typing. In the course of the type reconstruction of generic values, we may have free dynamic type variables which actually should not exist freely and must be instantiated to some closed type. The abstract type `dyn` is a good candidate of this closed type, since we can postpone the statically unsolved typing problems using the run time type checking mechanism of the dynamic values.

4.2.2 Type variables inside dynamic values — polymorphic contents

Using the dynamic encapsulation, types can be exported outside of their original scope. Therefore, the type information in dynamic values must be independent of any context. This means that dynamic type informations must be closed type schemes: all the type variables inside dynamic values must be quantified. Having type variables inside, dynamic values can encapsulate *polymorphic contents*. For instance, `dynamic []` will create a pair of (`[]`, α `list`) and it will be able to coerced back to a polymorphic list of type $\forall\alpha.\alpha$ `list`. To emphasize this, we write the type information of dynamic values as closed type schemes. For example, the value-type pair of the dynamic value of the polymorphic list is written as follows:

(`[]`, $\forall\alpha.\alpha$ `list`)

When a value v of type τ is encapsulated as a dynamic value $(v, \forall FV(\tau).\tau)$, we must check the generalizability of the type τ at type checking phase, since it is just a mono-type. If τ contains non-generalizable type variables, the encapsulation of $(v, \forall FV(\tau).\tau)$ must not be performed. Type checker must reject such encapsulation or automatically unify such non-generalizable type variables with some closed type. This new typing of the dynamized values is discussed in Section 4.3.

4.2.3 Dynamic primitives

The dynamic value primitives are easily implemented in extensional polymorphism framework. Their types are given as follows, with the abstract type for dynamic values, `dyn`:

```
type dyn (* the abstract type for dynamic values *)
dynamic : ∀δ.δ → dyn
coerce : ∀δ.dyn → δ
```

In the evaluation, the `dynamic` primitive receives the type of its argument as a part of its evaluation type context. On the other hand, the `coerce` receives the expected type of its coercion. They are compiled to functions[1] which receive these (run time) types as first arguments to create and destruct dynamic values:

$$\begin{array}{rcl} gen'(\texttt{dynamic}) & = & \texttt{fun t v} \ \rightarrow \ (\texttt{v},\texttt{t}) \\ gen'(\texttt{coerce}) & = & \texttt{fun t } (\texttt{v},\texttt{t}') \ \rightarrow \ \texttt{if is\_instance t t}' \texttt{ then v} \\ & & \qquad\qquad\qquad \texttt{else raise Coercion\_failure} \end{array}$$

Additional to the run time type argument, the compilation of `dynamic` also receives a value `v` and creates a value-type pair `(v,t)`. This pair is returned as a value of the abstract type `dyn`. `coerce` receives a dynamic value whose internal structure is `(v,t')` and returns its contents `v`, checking that the expected type `t` is a proper instance of `t'`. Otherwise, the coercion is rejected as a type violation. The predicate `is_instance` is equivalent to the predicate $\leq$ in our type system, whose definition is given by the pseudo code in Figure 4.1. The data types are identified simply by their names `t` at this moment.

[1]More precisely, these compilations are not the functions themselves, but their closures. For instance, $gen'(\texttt{dynamic}) = clos(\emptyset, \_, \texttt{t}, \texttt{fun v} \ \rightarrow \ (\texttt{v},\texttt{t}))$

```
let is_instance t t' =
  let subst = ref [] in
  let rec is_instance_aux t t' =
    match t, t' with
    | «τ», «α» →
        begin try
            if List.assoc «α» !subst = t then true else false
          with
            Not_found → subst := («α», t) :: !subst; true
        end
    | «τ1 → τ2», «τ1' → τ2'» →
        is_instance_aux «τ1» «τ1'» && is_instance_aux «τ2» «τ2'»
    | «τ1 * τ2», «τ1' * τ2'» →
        is_instance_aux «τ1» «τ1'» && is_instance_aux «τ2» «τ2'»
    | «base1», «base2» → base1 = base2
    | «(τ1,...,τn) t» «(τ1',...,τn'') t'» when n = n' && t = t' →
        is_instance_aux «τ1» «τ1'» && ... && is_instance_aux «τn» «τn'»
    | _ → false
  in
  is_instance_aux t t'
```

Figure 4.1: Definition of the predicate `is_instance`

Corresponding to the implicit generalization of type variables inside dynamic values, the stored run-time types are treated as closed type schemes in `is_instance`. The following relationship holds in fact:

`is_instance` $«\tau» «\tau'» \equiv \tau \leq \forall FV(\tau').\tau'$

Note that the dynamic primitives cannot be defined by `generic` constructs for the type-case based definition of generic values which will be introduced in Chapter 7, since generic values defined by the `generic` binding use dispatched run-time types just for branching and have no way to create values from received run time types.

4.3 Polymorphic encapsulation of dynamic values

As shown previously in the definition of the predicate `is_instance`, dynamic type informations are closed type schemes and all the type variables in them are quantified, so that it can be safe to create dynamic values with polymorphic contents. To ensure this property of value-type pairs, we must introduce new type checks for the dynamic primitives:

- At the encapsulation of v of type τ as $(v, \forall FV(\tau).\tau)$, the generalizability of τ must be checked. Otherwise non polymorphic values with free type variables could be encapsulated into dynamic values as polymorphic and break type safety at coercion.

- Polymorphic contents must be able to be coerced to values with polymorphic type schemes. This is impossible for the previous implementation of the `coerce` primitive as a generic primitive function. The coercion must be a special expression.

In the rest of this chapter, we study these problems of polymorphic encapsulation..

4.4 Polymorphic dynamization

Once type variables are encoded to run time types and captured into dynamic values, they can easily escape from their original scopes at typing. Therefore, the run time type variables inside dynamic values must be treated as closed type schemes which are independent of the typing contexts. This polymorphic encapsulation introduces implicit generalization of the dynamized expressions: the arguments to `dynamic` are silently generalized and stored as polymorphic inside dynamic values. We require special care in the type system so that only the type variables which are safely isolated from the typing context can be generalized and introduced into dynamic values. Otherwise, the type soundness is easily broken: for instance, suppose that `dynamic (ref [] :` α `list ref)` would create a dynamic value with a polymorphic content `(ref [],`$\forall\alpha.\alpha$ `list ref)` (which is clearly incorrect). We could insert anything to the reference cell and violates soundness by coercing this polymorphic content to corresponding types:

```
let d = dynamic (ref []) in
let x = coerce d : int  list ref in x := 1 :: !x;
let x = coerce d : bool list ref in x := true :: !x
(* the list now contains an integer and a boolean! *)
```

This problem can be avoided to restrict the type variable capturing only to the "*safely encapsulable*" type variables which are proven not to damage the type soundness. Most simply, if `dynamic` primitive takes values with non-encapsulable type variables, we should treat it as a typing error:

```
# dynamic (ref []);;
  ^^^^^^^
Error: dynamizer dynamic captures type α list ref,
  which contains a non-generalizable type variable.
```

This asks the programmer to eliminate the non-encapsulable type variables by adding an explicit type constraint without type variables. For instance:

```
# dynamic (ref [] : int list ref);;
- : dyn = <abstr> (* (ref [], int list ref) : this is safe *)
```

The type inference system can automate this elimination of dangerous non-encapsulable dynamic type variables, by unifying with some closed type. This ***dyn**-unification* is explained later in the type inference algorithm for the `dynamic` primitive, Section 4.4.5.

4.4.1 Encapsulable type variables

Now, our question is: what are the encapsulable type variables ?

Argument generalization check

In simple cases, when the run time types received by `dynamic` are statically known, encapsulable type variables are determined quite like the generalizable type variables in let bindings. For the expression `dynamic` $(e{:}\tau)$, we try to generalize the argument type τ implicitly, as if we had the following let binding around it:

```
dynamic (let v = (e:τ) in v)
```

The generalizable type variables in τ in this let binding are independent of the typing context of e, hence they are encapsulable. For example, in `dynamic (fun x → x : δ → δ)`, the type variable δ is encapsulable since it is generalized in `let v = fun x → x`. On the other hand, `dynamic (ref [] : δ list ref)` must not be well-typed and cannot create a polymorphic content since δ is not generalizable in the expression `let v = (ref [] : δ list ref)`.

The generalization checks for dynamization are restricted only against the arguments of the `dynamic` primitive. Making the places where the generalizations occur explicit as the let binding, we can keep the efficiency of the ML type inference. This restriction has, however, a drawback: the expressions which are not arguments of the `dynamic` primitive cannot be encapsulated polymorphically. The following example is rejected to encapsulate the identity function as polymorphic, though it is completely safe:

```
# (fun x → dynamic x) (fun x → x)
            ^^^^^^^
Error: dynamic captures type α → α
  which contains a non-generalizable type variable.
```

Derived dynamizers

The argument generalization of the `dynamic` primitive can be applied only to the type variables which are not generalized by any outer polymorphism. The type variables generalized outside must be treated differently, otherwise any *derived dynamizer*, derived generic values which use `dynamic` inside, would be ill-typed:

```
let double_dynamic = function x → dynamic (x, x) in ...
```

In the above definition of the derived dynamizer `double_dynamic`, the argument of the `dynamic` primitive `(x, x)` has the type $\delta * \delta$, where δ is bound to the variable `x` and not generalizable at the position of `(x, x)`. If we would apply the argument generalization to the type variable δ, it would be regarded as non-encapsulable and therefore had to be eliminated: `double_dynamic` would be forced to be monomorphic. This is too strict and not what we want.

We must not perform the type generalization check of the type variables which are generalized outside, since they are not the actual type which are encapsulated into dynamic values; they are instantiated to other types at run time, depending on how the derived dynamizers are used. For example, if in an application `double_dynamic` e, e has a type τ, the expression `(x, x)` is instantiated to $\tau * \tau$ in its evaluation. Whether the type variables in $\tau * \tau$ are encapsulable or not should not be determined by the generalization of `(x, x)` but by the type generalization of the expression $e : \tau$.

Can we find the encapsulated type variables of the derived dynamizers and secure the polymorphic encapsulation by their argument generalization check, just like the `dynamic` primitive? Unfortunately not. The following example shows that we need more typing analysis:

```
let apply_and_dynamize g =
  let v = g () in
  dynamic v
in
apply_and_dynamize (fun () → ref [])
```

The type of the argument `(fun () → ref [])` itself is generalizable to $\forall\delta.$`unit` $\rightarrow \delta$ `list ref`, but the derived dynamizer `apply_and_dynamize` creates a dynamic value of `ref []` which cannot be polymorphic: though δ is generalizable at the use of the derived dynamizer, it is not encapsulable. In order to assure the security of polymorphic encapsulation of the derived dynamizers, their definitions must be investigates. This additional type checking may affect the efficiency of the type inference (See Section 4.6).

Here we propose a restricted, but simpler solution. We give up the polymorphic encapsulation of derived dynamizers: only the direct arguments to the `dynamic` primitive can be polymorphically encapsulated into dynamic values. Our solution is summarized as follows:

The `dynamic` primitive

- Only the `dynamic` primitive can encapsulate the type variables of its arguments as quantified, after their generalization checks. For instance, `dynamic (fun x → x)` creates a dynamic value `(fun x → x,` $\forall\alpha.\alpha \rightarrow \alpha$`)`.
- The type variables generalized outside, which defines derived dynamizers are kept well-typed. (However, their instantiations at run time are restricted to closed types.): `double_dynamic` can have a polymorphic type scheme $\forall\delta.\delta \rightarrow$ `dynamic`.
- The other type variables annotated to the arguments of the `dynamic` primitive are rejected, since they are not encapsulable.

Derived dynamizers

- The definition of the derived polymorphic dynamizers using `dynamic` or other derived dynamizers is not restricted at all.
- Their arguments cannot have any type variable in their type annotations, unless they are generalized outside. For example, `double_dynamic (fun x → x)` is well-typed only when the expression `(fun x → x)` has a closed type.

4.4.2 Typing rules for the polymorphic dynamization

We need to distinguish the derived dynamizers from the other generic values by looking at their type schemes so that our additional typing restriction can be applied only to the arguments of the derived dynamizers. The current GML type system expressiveness does not tell whether a generic value is a dynamizer or not.

Let us introduce, temporarily, a new typing system, for discussing the safety of the polymorphic dynamization. Once we introduce the generic constraint type system in Chapter 8, all the result of this section will be easily integrated. In addition, with generic constraints we will not need the distinction of the following poly-dynamic and normal dynamic type variables. The poly-dynamic quantifications in this formalizations will be translated to normal dynamic variable quantifications with derived constraints of the `dynamic` primitive. Look at Section 4.7 further details of the type system integration.

Poly-dynamic type variables and partial quantification

To distinguish normal generic values and dynamizers, we introduce another special subset of dynamic type variables called *poly-dynamic type variables* $\mathbb{P} \subset \mathbb{D}$ whose elements are written as $\partial, \partial_1, \partial_2, \ldots$. Now the type schemes of the dynamizers are quantified by poly-dynamic type variables. For example,

$$\texttt{dynamic} : \forall \partial . \partial \rightarrow \texttt{dyn}$$

Poly-dynamic type variables are dynamic type variables with a special restriction: the existence of non-quantified poly-dynamic type variables ∂ in the well-typed expressions is not allowed, unless they are generalized by some generalization, `let` (and `generic` bindings which will be introduced later), or the implicit generalization of the direct argument of `dynamic`. This "poly-" attribute of the dynamic type variables is propagated by substitution during type checking, by redefining the notion of the admissible substitutions:

Definition 8 (Redefined admissible substitutions)
A substitution S is admissible if

$$\forall \delta \in Dom(S),\ \forall \nu \in FV(S(\delta)),\ \nu \in \mathbb{D}$$
$$\forall \partial \in Dom(S),\ \forall \nu \in FV(S(\partial)),\ \nu \in \mathbb{P}$$

Admissible substitutions map poly-dynamic type variables to expressions without regular and non poly-dynamic type variables.

We also introduce a notation $PV(\tau)$ and $PV(e_t)$, for free poly-dynamic type variables of a type τ and an expression e_t, given by the following definitions:

$$PV(\tau) = DV(\tau) \cap \mathbb{P}$$
$$PV(e_t) = DV(e_t) \cap \mathbb{P}$$
$$PV(\sigma) = FV(\sigma) \cap \mathbb{P}$$

Instantiations of the poly-dynamic quantifications of derived dynamizers are more restricted than the `dynamic` primitive. The type variables of the instances must be generalized by some outer polymorphisms, since the derived dynamizers lack the argument generalization. To emphasize this rule, we put an additional *partial* attribute $\underline{\partial}$ to poly-dynamic quantifications of the derived dynamizers:

$$\texttt{double\_dynamic : } \forall \underline{\partial}.\partial \to \texttt{dyn}$$

The instance relationship $\leq$ of types and type schemes is as same as before, even with this partial quantification attribute:

$$\tau' \leq_S \forall V.\tau \equiv S(\tau) = \tau',\ Dom(S) \subset V \text{ ('\_' signes in } V \text{ are ignored)}$$

We introduce a notation $\underline{V}$, the partialization of the set of variables V, whose definition is given as follows. This adds the partial quantification attribute to all the poly-dynamic variables in V:

$$\underline{V} = \{\underline{\partial} \mid \partial \in V\} \cup (V \setminus \mathbb{P})$$

At this moment, this partial quantification does not play a big role actually: the poly-dynamic quantifications of the derived dynamizers must be *always partial*. But later, when we relax the instantiation restriction of the derived dynamizers, they will be able to have fully quantified poly-dynamic variables, too.

Typing rules with poly-dynamic type variables

With this distinction between poly-dynamic type variables for dynamizers and non-poly variables for the other generic values, we can build a new type system which can reject non-encapsulable dynamizations. The new typing judgments take the following form: $\Gamma, P \vdash e_t$, where P is the set of poly-dynamic type variables called *generalized poly-dynamic variable set*. This P indicates which poly-dynamic variables can exist in e_t. Its elements must be generalized some generalization rules. The instantiation rule is modified to reflect this:

$$\textbf{(pt-inst)}\ \frac{\Gamma(\texttt{x}) = \sigma, \quad \tau \leq \sigma, \quad PV(\tau) \subset P}{\Gamma, P \vdash \texttt{x}_\sigma : \tau}$$

If variable `x` has a type τ, the poly-dynamic type variables in τ must be members of P.

The generalization rules extend generalized poly-dynamic variable set P with P', and type check their generalization target expressions. The quantified poly-dynamic variables in V must be same as the added poly-dynamic variables P'. The poly-dynamic quantifications must be partialized in the type environment for the definitions of the derived dynamizers.

$$\textbf{(pt-let)}\ \frac{\begin{array}{c}\Gamma, P \cup P' \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \quad P' = V \cap \mathbb{P} \\ \Gamma\{\texttt{x} : \forall \underline{V}.\tau_1\}, P \vdash e_{t2} : \tau_2\end{array}}{\Gamma, P \vdash \texttt{let x}_{\underline{V}}\texttt{ = }e_{t1}\texttt{ in }e_{t2} : \tau_2}$$

$$\textbf{(pt-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{f}:\tau_1\}, P \cup P' \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1) \\ P' = V \cap \mathbb{P} \quad \Gamma\{\mathtt{f} : \forall \underline{V}.\tau_1\}, P \vdash e_{t2} : \tau_2\end{array}}{\Gamma, P \vdash \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2}$$

We have another generalization rule for the application of the `dynamic` generic primitive. In this time, the quantified type variables of the generalized type scheme of the argument e_t must be as same as the extended set P':

$$\textbf{(pt-dynamic)}\ \frac{\Gamma, P \cup P' \vdash e_t : \tau \quad \forall P'.\tau = Gen(\Gamma, e_t : \tau) \quad P' \subset \mathbb{P}}{\Gamma, P \vdash ((\mathtt{dynamic}_{\forall \partial.\partial \rightarrow \mathtt{dyn},\ P'} : \tau \rightarrow \mathtt{dyn})\ (e_t : \tau)) : \mathtt{dyn}}$$

An instance of the `dynamic` primitive takes an additional type annotation P', the generalized variables of its argument, like the generalized variables annotations of `let` pattern variables. Like the let bindings, this generalized poly-dynamic variables bind their occurrences in e_t:

$$PV((\mathtt{dynamic}_{\forall \partial.\partial \rightarrow \mathtt{dyn},\ P} : \tau \rightarrow \mathtt{dyn})\ (e_t : \tau)) \quad = \quad PV(\tau) \setminus P$$

The other typing rules are in Figure 4.2. The rules for constants and generic primitives restrict poly-dynamic variables within their generalized poly-dynamic variable sets like (pt-inst). The other typing rules are same as before, just with the additions of generalized poly-dynamic variable sets.

$$\textbf{(pt-cnst)}\ \frac{\tau \leq \sigma \quad PV(\tau) \subset P}{\Gamma, P \vdash c_\sigma : \tau} \qquad \textbf{(pt-gen)}\ \frac{\tau \leq \sigma \quad PV(\tau) \subset P}{\Gamma, P \vdash \mathtt{g}_\sigma : \tau}$$

$$\textbf{(pt-fun)}\ \frac{\Gamma\{\mathtt{x} : \tau_1\}, P \vdash e_t : \tau_2}{\Gamma, P \vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2} \qquad \textbf{(pt-tpl)}\ \frac{\Gamma, P \vdash e_{t1} : \tau_1 \quad \Gamma, P \vdash e_{t2} : \tau_2}{\Gamma, P \vdash (e_{t1}, e_{t2}) : \tau_1 * \tau_2}$$

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

Figure 4.2: The other typing rules

A program e_t is well-typed if $\emptyset, \emptyset \vdash e_t$ holds: all the poly-dynamic type variables introduced in generalized poly-dynamic variable set must be eliminated by generalization rules.

Now let's see how this type system provides the type safety of the polymorphic dynamizations: the expression `dynamic (fun x` $\rightarrow$ `x)` has the following typing, which checks that the arguments can be generalizable to $\forall \partial.\partial \rightarrow \partial$. This proves that `fun x` $\rightarrow$ `x` is safely dynamizable as `(fun x` $\rightarrow$ `x,`$\forall \alpha.\alpha \rightarrow \alpha$`)` :

$$\frac{\emptyset, \{\partial\} \vdash \mathtt{fun\ x} \rightarrow \mathtt{x} : \partial \rightarrow \partial \quad \forall \partial.\partial \rightarrow \partial = Gen(\emptyset, \mathtt{fun\ x} \rightarrow \mathtt{x} : \partial \rightarrow \partial)}{\emptyset, \emptyset \vdash \mathtt{dynamic}\ (\mathtt{fun\ x} \rightarrow \mathtt{x}) : \mathtt{dyn}}$$

On the other hand, if the argument of dynamic cannot be fully generalized, (pt-dynamic) cannot eliminate the ungeneralizable poly-dynamic variables from the set P. For example:

$$\dfrac{\dfrac{\{\mathtt{x} : \partial\}, \{\partial\} \vdash (\mathtt{x}, \mathtt{x}) : \partial * \partial \qquad \partial * \partial = Gen(\{\mathtt{x} : \partial\}, (\mathtt{x}, \mathtt{x}) : \partial * \partial)}{\{\mathtt{x} : \partial\}, \{\partial\} \vdash \mathtt{dynamic}\ (\mathtt{x}, \mathtt{x}) : \mathtt{dyn}}}{\emptyset, \{\partial\} \vdash (\mathtt{fun\ x} \rightarrow \mathtt{dynamic}\ (\mathtt{x}, \mathtt{x})) : \partial \rightarrow \mathtt{dyn}}$$

In the above, the variable ∂ is not generalizable because it appears in the type environment $\{\mathtt{x} : \partial\}$. It remains in the generalized poly-dynamic variable set and must be eliminated by some generalization, in order to make this expression as a part of well-typed program. This is done by using the expression as a definition of a derived dynamizer:

$$\dfrac{\begin{array}{c}\emptyset, \{\partial\} \vdash (\mathtt{fun\ x} \rightarrow \mathtt{dynamic}\ (\mathtt{x}, \mathtt{x})) : \partial \rightarrow \mathtt{dyn} \\ \forall \partial . \partial \rightarrow \mathtt{dyn} = Gen(\emptyset, (\mathtt{fun\ x} \rightarrow \mathtt{dynamic}\ (\mathtt{x}, \mathtt{x})) : \partial \rightarrow \mathtt{dyn}) \\ \{\mathtt{double\_dynamic} : \forall \underline{\partial} . \partial \rightarrow \mathtt{dyn}\}, \emptyset \vdash e_t : \tau\end{array}}{\emptyset, \emptyset \vdash \mathtt{let\ double\_dynamic = fun\ x} \rightarrow \mathtt{dynamic}\ (\mathtt{x}, \mathtt{x})\ \mathtt{in}\ e_t : \tau}$$

By the restriction of (pt-inst), the partially quantified poly-dynamic type variable of double_dynamic must be instantiated to closed types, unless it is used to define another derived dynamizer. For instance the following application of (pt-app) rule is valid only for closed types τ:

$$\dfrac{\begin{array}{c}\{\mathtt{double\_dynamic} : \forall \underline{\partial} . \partial \rightarrow \mathtt{dyn}\}, \emptyset \vdash \mathtt{double\_dynamic} : \tau \rightarrow \mathtt{dyn} \\ \{\mathtt{double\_dynamic} : \forall \underline{\partial} . \partial \rightarrow \mathtt{dyn}\}, \emptyset \vdash \mathtt{fun\ x} \rightarrow \mathtt{x} : \tau \rightarrow \tau\end{array}}{\{\mathtt{double\_dynamic} : \forall \underline{\partial} . \partial \rightarrow \mathtt{dyn}\}, \emptyset \vdash (\mathtt{double\_dynamic}\ (\mathtt{fun\ x} \rightarrow \mathtt{x})) : \mathtt{dyn}}$$

4.4.3 Type safety of the polymorphic dynamization

We here try to prove the type safety of the polymorphic dynamization under the type system defined in the previous section. Suppose we have a well-typed program and inside its typing we have the following use of (pt-dynamic):

$$\dfrac{\Gamma, P \cup P' \vdash e_t : \tau \qquad \forall P' . \tau = Gen(\Gamma, e_t : \tau) \qquad P' \subset \mathbb{P}}{\Gamma, P \vdash (\mathtt{dynamic}_{\forall \partial . \partial \rightarrow \mathtt{dyn}} : \tau \rightarrow \mathtt{dyn})\ (e_t : \tau) : \mathtt{dyn}}$$

the safety of the encapsulation of the poly-dynamic variables of P' are clear, since they are generalized in the rule. The uneliminated poly-dynamic variables of P must be all generalized at the definition of derived dynamizers, by (pt-let) or (pt-letrec), and at the evaluation of the expression they must be instantiated to closed types, since their instantiations are restricted by (pt-inst). Before the formal proof, we need a substitution lemma of the new typing system and make the evaluation system up-to-date.

Lemma 11 (Substitution lemma of typing) *Let Γ be a typing environment, P_0 and P be distinct sets of poly-dynamic type variables, e_t be a fully type-annotated GML expression and S be a substitution where $Dom(S) \cap \mathbb{P} = P$ and $\forall \nu \in Dom(S), PV(S(\nu)) = \emptyset$.*

Suppose e_t has a typing $\Gamma, P_0 + P \vdash e_t$. Then,

$$S(\Gamma), P_0 \vdash S(e_t)$$

PROOF: Proof is omitted. See the corresponding lemma in the generic constraint type system, Lemma 32. □

Lemma 12 *If an expression e_t has a typing $\Gamma, P \vdash e_t : \tau$, $PV(\tau) \subset P$.*

PROOF: It is clear by induction on the typing derivation. □

Evaluation system update

We update the evaluation system so that they can also have poly-dynamic variables in their type annotations. Some of the semantic typing rules previously defined in Section 2.5.5 are also to be extended as follows:

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

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

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

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

The substitution lemma of the semantic typing, Lemma 8 is also expanded for this new evaluation system:

Lemma 13 (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, we have $S(E) \in S(\Gamma)$.*

PROOF: The proof is almost as same as one for Lemma 8. □

The safety of the polymorphic dynamization is proven by showing that the type variables encapsulated in dynamic values are poly-dynamic type variables only which are generalized in (pt-dynamic). The other poly-dynamic type variables are instantiated to closed types in evaluation:

Theorem 14 (Safety of polymorphic dynamization) *Let e_t be a well-typed program with a typing $\Gamma, \emptyset \vdash e_t$. Let E be an evaluation environment such that $E \in \Gamma$, then in the evaluation of $E \models e_t \Longrightarrow r$, for any sub-evaluation $E' \models e'_t : \tau' \Longrightarrow r'$, we have $PV(\tau') = \emptyset$ except inside the evaluation of dynamic primitive applications* `dynamic` e'_t.

If a dynamic primitive application `dynamic` $(e'_t : \tau')$ *is evaluated, there exists a typing environment Γ' and a set of poly-dynamic type variables P such that $\Gamma', P \vdash e'_t : \tau'$ and $Gen(\Gamma', e'_t : \tau') = \forall P.\tau'$.*

PROOF: Proven by induction on the evaluation steps.

Variables Let $e_t = \mathtt{x}$ and its typing be $\Gamma, \emptyset \vdash \mathtt{x}_\sigma : \tau$ for some σ and τ. From Lemma 12, $PV(\tau) = \emptyset$.

There are two applicable rule groups for variables depending on which kind of semantic values are bound to them — (e-nvar) for regular values and (e-gvar-*) for generic closures (from the type soundness, the rules for errors are never used). If we use (e-nvar):

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

We do not require any sub-evaluation therefore we proven its case.

If we use (e-gvar):

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

The evaluation requires another evaluation $E' \models S(e_t : \tau') \Longrightarrow r$, where $S(\tau') = \tau$ and $PV(\tau) = \emptyset$. From the updated semantic typing rule (s-gclos), we obtain the typing for e'_t, for some type environment Γ':

$$E' \in \Gamma' \qquad \Gamma', \underline{V} \vdash e'_t : \tau' \qquad \forall V.\tau' = Gen(\Gamma', e'_t : \tau') \qquad V \cap \mathbb{D} \neq \emptyset$$

Since $Dom(S) = V$, from Lemma 11, $S(\Gamma'), \emptyset \vdash S(e'_t)$. From Lemma 13, $S(E') \in S(\Gamma')$. The correctness of the theorem for the sub-evaluation $E' \models S(e_t : \tau') \Longrightarrow r$ is clear from the induction hypothesis.

The case (e-gvar-r) is applied is also proven in the same way.

Constants and generic primitives They are proven like the case of variables.

Recursive let bindings Let $e_t = \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2$, then its typing is obtained from (pt-letrec):

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

Since τ_1 is a type for `fun x` $\rightarrow e_{t1}$, it must be a functional type and we let $\tau_1 = \tau_{11} \rightarrow \tau_{12}$. From Lemma 12, $PV(\tau_2) = \emptyset$.

Like the case of the variables we have two possibility of evaluation. If the quantified variables V contains no dynamic variable, this is a `let`-definition of non-generic value and (e-letrec) is used, otherwise the binding defines a generic value and (e-g-letrec) is applied:

$$\textsf{(e-letrec)}\quad \frac{V \cap \mathbb{D} = \emptyset \qquad E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1}))\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

$$\textsf{(e-g-letrec)}\quad \frac{V \cap \mathbb{D} \neq \emptyset \qquad E\{\mathtt{x} = gclos(E, \mathtt{x}, V, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1}))\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec\ x}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

No matter which rule is used, its subsequent evaluation is for $e_{t2} : \tau_2$, where $PV(\tau_2) = \emptyset$. The theorem is correct for this sub evaluation of e_{t2} from the induction hypothesis.

Let bindings The possible evaluation rules are (e-let) and (e-g-let). The case of (e-g-let) is proven similarly to the case of recursive let bindings, since it stores its definition as a generic closure without evaluation. In the case of (e-let), the definition expression is also evaluated, but its type does not contain dynamic variables. Therefore the theorem is also correct for the evaluation of the definition expression, using the induction hypothesis.

Applications of `dynamic` primitive This is the exceptional case of the theorem. The correctness of the case is clear from its typing:

$$\frac{\Gamma, P \vdash e'_t : \tau' \qquad \forall P.\tau' = Gen(\Gamma, e'_t : \tau') \qquad P \subset \mathbb{P}}{\Gamma, \emptyset \vdash ((\mathtt{dynamic}_{\forall \partial.\partial \rightarrow \mathtt{dyn}} : \tau' \rightarrow \mathtt{dyn})\ (e'_t : \tau')) : \mathtt{dyn}}$$

The others Their corresponding typing rules do not modify generalized poly-dynamic type variable sets.

□

4.4.4 Type reconstruction in two phases

Our new type system introduced a new restriction: the poly-dynamic type variables must be generalized somehow, by let binding or `dynamic` argument generalizations. Under this restriction, dynamic value related expressions may lose principal typing:

```
let double_dynamic x = dynamic (x,x) in
double_dynamic [];;
```

In this example, the use of `double_dynamic` can have types like `int list` $\rightarrow$ `dyn`, `string list` $\rightarrow$ `dyn`, etc..., but not ∂ `list` $\rightarrow$ `dyn`, which could be the principal typing of `double_dynamic` in the type system without the restriction.

Type reconstruction without the principal typing property always faces to ambiguity problems. Expressions can have more than one possible typings which are incompatible each other. Usually type checkers just reject these ambiguities as typing errors or choose one of the typings as default. We do the same: first we define a reconstruction algorithm which rejects the ambiguity dynamizers as type errors.

Our type reconstruction consists of two phases. The first phase is the usual ML type reconstruction with the poly-dynamic variables. The result of this phase may contain illegal dynamizer instances which have unencapsulable poly-type variables.

The second phase is the handling of these unencapsulable type variables. At first we just reject them as type errors. But later we propose an automatic ambiguity resolution using the abstract type `dyn` (Section 4.4.5).

First phase

The algorithm of the first phase W_1 whose definition is in Figure 4.3 is obtained as a simple extension of the original W (Figure 7.2 of Section 2.6). First, to propagate the attribute of poly-dynamic type variables by unification, we define a function *polydyn*. Its definition is like *dynamize* function:

$$
\begin{aligned}
\mathit{polydyn}(\mathit{base}) &= \emptyset \\
\mathit{polydyn}(\alpha) &= \{\alpha \Rightarrow \partial\} \text{ where } \partial \text{ is a fresh poly-dynamic type variable} \\
\mathit{polydyn}(\delta) &= \{\delta \Rightarrow \partial\} \text{ where } \partial \text{ is a fresh poly-dynamic type variable} \\
\mathit{polydyn}(\partial) &= \emptyset \\
\mathit{polydyn}(\tau_1 \rightarrow \tau_2) &= \text{let } S_1 = \mathit{polydyn}(\tau_1),\ S_2 = \mathit{polydyn}(S_1(\tau_1)) \text{ in } S_2 \circ S_1 \\
\mathit{polydyn}(\tau_1 * \tau_2) &= \text{let } S_1 = \mathit{polydyn}(\tau_1),\ S_2 = \mathit{polydyn}(S_1(\tau_1)) \text{ in } S_2 \circ S_1
\end{aligned}
$$

The unification algorithm *unify* is extended for the case of poly-dynamic type variable, using *polydyn* function:

$$
\mathit{unify}(\partial, \tau) \quad = \quad S \circ \{\partial \Rightarrow S(\tau)\} \text{ when } \partial \notin FV(\tau), \text{ where } S = \mathit{polydyn}(\tau)
$$

The new case for the `dynamic` primitive is added to the algorithm W_1. The type of argument of `dynamic` is poly-dynamized by unifying with ∂, then generalized. The generalized poly-dynamic variables is attached to the type annotation of `dynamic`. The cases for let bindings are slightly modified so that they partially quantify poly-dynamic variables of the derived dynamizers. The other cases are kept as they are.

$$
\begin{array}{lcl}
W_1(\Gamma, \texttt{dynamic}\ e) & = & \text{let } e_t : \tau = W_1(\Gamma, e) \\
 & & \text{let } S = \mathit{unify}(\partial, \tau) \text{ where } \partial \text{ is fresh} \\
 & & \text{let } \sigma = \mathit{Gen}(S(\Gamma), S(e_t)) \\
 & & \text{in } ((\texttt{dynamic}_{\forall\partial.\partial \,\to\, \texttt{dyn},\ BPV(\sigma)} : S(\tau) \to \texttt{dyn})\ S(e_t)) : \texttt{dyn},\ S
\end{array}
$$

$$
\begin{array}{lcl}
W_1(\Gamma, \texttt{let x} = e_1 \texttt{ in } e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W_1(\Gamma,\ e_1) \\
 & & \text{let } \forall V.\tau_1 = \mathit{Gen}(S_1(\Gamma), (e_{t1} : \tau_1)) \\
 & & \text{let } e_{t2} : \tau_2,\ S_2 = W_1(S_1(\Gamma)\{\texttt{x} : \forall \underline{V}.\tau_1\},\ e_2) \\
 & & \text{in } \texttt{let x}_{\underline{V}} = S_2(e_{t1}) \texttt{ in } e_{t2} : \tau_2,\ S_2 \circ S_1
\end{array}
$$

$$
\begin{array}{l}
W_1(\Gamma, \texttt{let rec f} = \texttt{fun x} \to e_1 \texttt{ in } e_2) = \\
\qquad \text{let } \alpha \text{ be a fresh variable} \\
\qquad \text{let } (\texttt{fun x} \to e_{t1}) : \tau_1,\ S_1 = W_1(\Gamma\{\texttt{f} : \alpha\},\ \texttt{fun x} \to e_1) \\
\qquad \text{let } S'_1 = \mathit{unify}(\tau_1, S_1(\alpha)) \\
\qquad \text{let } \forall V.\tau'_1 = \mathit{Gen}((S'_1 \circ S_1)(\Gamma), S'_1(\texttt{fun x} \to e_{t1} : \tau_1)) \\
\qquad \text{let } e_{t2} : \tau_2,\ S_2 = W_1(((S'_1 \circ S_1)(\Gamma))\{\texttt{f} : \forall \underline{V}.\tau'_1\},\ e_2) \\
\qquad \text{in } \ \texttt{let rec f}_{\underline{V}} = (S_2 \circ S'_1)(\texttt{fun x} \to e_{t1}) \\
\qquad \qquad \texttt{in } e_{t2} : \tau_2,\ S_2 \circ S'_1 \circ S_1
\end{array}
$$

For the other cases, $W_1(\Gamma, e)$ is obtained from $W(\Gamma, e)$, replacing all the recursive occurrences of W by W_1.

Figure 4.3: Type reconstruction algorithm for the polymorphic dynamization

Second phase

Once the fully type-annotated program e_t is obtained from the untyped source e as $e_t, S = W_1(\emptyset, e)$, The second reconstruction W_2 checks the existence of the free poly-dynamic type variables. If it is not an empty set, we reject e_t as ill-typed:

$$W_2(e_t) = \ \text{if } PV(e_t) = \emptyset \text{ then } e_t \text{ else reject } e_t \text{ as a type error}$$

For better user interface, the type checker should point out the problematic dynamizer in its error message:

```
# dynamic (ref []);;
  ^^^^^^^
Error: dynamizer dynamic captures type _α list ref,
  which contains a non-generalizable type variable.
```

Algorithm correctness

Now we prove the correctness and partial completeness of the algorithm. They can be proven in the same way as we have done in Section 2.6:

Theorem 15 (Type reconstruction correctness) *The algorithm W_1 is correct with respect to the typing rules. Let Γ be a typing environment and e be a bare GML expression and if $W_1(\Gamma, e) = e_t,\ S$, then $S(\Gamma), PV(e_t) \vdash e_t$.*

PROOF: (Sketch) Proven by induction over the structure of bare GML expression, see the proof of the Theorem 9 for details. □

From Theorem 15, we obtain the following immediately:

Theorem 16 (Type reconstruction correctness for programs) *The algorithm W_1 and W_2 are correct with respect to the typing rules. Let e be a bare GML program and if $W_1(\emptyset, e) = e_t,\ S$ and $W_2(e_t)$ does not reject e_t, then $\emptyset, \emptyset \vdash e_t$.*

PROOF: From $W_2(e_t) = e_t$, we have $PV(e_t) = \emptyset$. Using Theorem 15, $\emptyset, \emptyset \vdash e_t$. □

The type reconstruction cannot be complete, since the new type system lacks principal typing. But we can prove a weak form of completeness:

Theorem 17 (Type reconstruction weak completeness) *Let Γ be a typing environment, $e'_t : \tau'$ be a GML expression, and V be a set of poly-dynamic type variables. If there exists a typing $S'(\Gamma), V \vdash e'_t : \tau'$ where $Dom(S') \subset FV(\Gamma)$, then $W_1(\Gamma, strip(e'_t)) = e_t : \tau,\ S$ and there exists a substitution S'' such that*

$$S''(\tau) = \tau', \quad \forall \nu \notin fresh(W(\Gamma, strip(e'_t))), \quad S'(\nu) = S'' \circ S(\nu)$$

In the latter statement, $fresh(W_1(\Gamma, strip(e'_t)))$ denotes the set of newly introduced type variables in W_1.

PROOF: (Sketch) By induction over the structure of the bare GML expression. The proof follows the same outline of the one of Theorem 17. The poly-dynamic variable set V does not play any important role in the proof. □

With the second phase W_2, Theorem 17 is rewritten as follows for programs:

Theorem 18 (Type reconstruction weak completeness for programs) *Let $e'_t : \tau'$ be a GML expression and $\emptyset, \emptyset \vdash e'_t : \tau'$ be its typing. If $W_1(\emptyset, strip(e'_t)) = e_t : \tau, S$ and $W_2(e_t)$ does not reject e_t, there is a substitution S' such that $S'(\tau) = \tau'$.*

PROOF: This is a special case of Theorem 17. □

Theorem 18 indicates that when W_1 and W_2 succeed, they always return the principal typings.

4.4.5 `dyn`-unification

The reconstruction algorithm W_1 may return too general and invalid typing which contains unencapsulable poly-dynamic type variables. We can easily obtain valid typing from them, by unifying these unencapsulable type variables with some closed type expressions. However, W_2 does not try it but rejects these type variables as typing errors. In some sense it is a correct behavior, since such expressions lack principal typings. Since possible typings are incompatible each other, the expression typing is ambiguous: even if the type reconstruction selects one of them, it may be completely different from the programmer's intention. For example, to be well-typed in the following expression

```
let x = ref [] in dynamic x;;
```

`x` must have a type expression τ `list ref` for some close type τ. τ can be `unit`, `int` , etc.... None of them have stronger reason to be chosen as the unification candidate than others; once unified, `x` becomes monomorphic and we cannot use it in the other types.

There is an exception. The abstract type `dyn`. Of course even this `dyn` cannot prevent `x` from being monomorphic, but `dyn` works like a polymorphic type variable, thanks to its pseudo heterogeneousness. Any type can be hidden and encapsulated into this type using the `dynamic` primitive:

```
# let x = ref [] in dynamic x;;
- : dyn
# x;; (* x's type variable is unified automatically with dyn *)
- : dyn list ref = {contents = []}
# x := dynamic 1 :: !x;;
# x := dynamic "hello" :: !x;;
...
```

We can regard the type `dyn` is the most "general" type of the closed type expressions. From this generality, we can use it as the type of the auto ambiguous resolution. We call this ***dyn*-*unification*** of unencapsulable type variables. We will also use this `dyn`-unification later, in the final ambiguity resolution of user-defined generic values (Section 10.4.1).

Once the fully type-annotated expression e_t is reconstructed by the algorithm W_1, the new second phase W_2' unifies the free occurrences of poly-dynamic type variables with with `dyn`:

$$W_2'(e_t) = S(e_t) \text{ where } S = \{(\partial \Rightarrow \texttt{dyn}) \mid \forall \partial, \partial \in PV(e_t)\}$$

The well-typedness of $S(e_t)$ is preserved thanks to Theorem 3. Since W_2' never rejects expressions, now the type reconstruction algorithm W_1 and W_2' always succeed if given expressions are typable.

Printing warnings

In spite of the generality of the type `dyn`, the auto unification of unencapsulable type variables may produce incompatible typing from the user intention and be confusing. To have a better interface, the type reconstruction algorithm should report the `dyn`-unifications as warnings:

```
let x = ref [] in dynamic x;;
Warning: the unencapsulable type variable of x is unified with dyn
- : dyn
```

4.5 Polymorphic coercion

The implementation of coercion as the generic primitive function `coerce` was simple, but it has a critical problem: as far as it is a generic primitive, the coercion of the dynamic values must take a form of application `coerce` e_t, which is expansive and therefore not generalizable under the value polymorphism restriction[Wri95]. This means that the polymorphic contents of dynamic values cannot be coerced polymorphically. For example, the dynamic value of the polymorphic identity function is coerced to just a monomorphic function:

```
# let d = dynamic (fun x → x);;
d : dyn = <abstr> (* = (<fun>, ∀α.α → α) *)
# let f = (coerce d : α → α);;
f : _α → _α = <fun>
```

To coerce back polymorphic contents to polymorphic values, we need to make the application `coerce` e_t exceptionally generalizable, or have a new *coerce expression* which looks like the pattern match expression:

$$\texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x}{:}\sigma\ \Rightarrow\ e'_t$$

A `coerce` expression $\texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x}{:}\sigma\ \Rightarrow\ e'_t$ takes a dynamic value e_t and checks whether its content can be coerced to the explicitly written type scheme σ, which is closed and without dynamic type variables. If it is, the content of the dynamic value is bound to the pattern variable `x` and e'_t is evaluated. Otherwise it raises a run time exception. The typing rule for this coerce expression is straightforward: the value to be coerced must be a dynamic value, and the result of the whole expression is the type of the e'_t:

$$\frac{\Gamma \vdash e_t : \texttt{dyn} \quad \Gamma\{\texttt{x} : \sigma\} \vdash e'_t : \tau \quad FV(\sigma) = \emptyset \quad BDV(\sigma) = \emptyset}{\Gamma \vdash \texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x} : \sigma \Rightarrow e'_t : \tau}$$

The compilation of the coerce expression is almost same as the original `coerce` generic primitive:

$$\lfloor \texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x} : \forall V'.\tau \Rightarrow e'_t \rfloor_V = \begin{array}{l}\texttt{let (v,ty) = } e_t \texttt{ in} \\ \texttt{if is\_instance } \langle\!\langle \tau \rangle\!\rangle \texttt{ ty then} \\ \quad \texttt{(fun x} \rightarrow \lfloor e'_t \rfloor_V\texttt{) v} \\ \texttt{else raise Coercion\_failure}\end{array}$$

The `coerce` expression provides polymorphic coercion since the content which is bound to the pattern variable `x` has a possibly polymorphic type scheme σ. The following code which uses the content polymorphically is well-typed and correctly executed:

$$\texttt{coerce (dynamic (fun x} \rightarrow \texttt{x)) with f:}\forall\alpha.\alpha \rightarrow \alpha \Rightarrow \texttt{f 1; f true}$$

Combining pattern match expression, this `coerce` expression can be extended to have multiple cases with more complex patterns, like `typecase` expression in [ACPP91]:

$$\begin{array}{l}\texttt{coerce } e_{t1}\texttt{,...,}e_{tn} \texttt{ with} \\ \texttt{| (x}_{11} \texttt{ : } \tau_{11}\texttt{),..., (x}_{1n} \texttt{ : } \tau_{1n}\texttt{)} \rightarrow e'_{t1} \\ \texttt{| ...} \\ \texttt{| (x}_{m1} \texttt{ : } \tau_{m1}\texttt{),..., (x}_{mn} \texttt{ : } \tau_{mn}\texttt{)} \rightarrow e'_{tm}\end{array}$$

4.6 Future work: relaxing the restriction

We studied the type checking to ensure the safety of polymorphic dynamization, but our typing may be too restricted in practice: the derived dynamizers cannot create polymorphic encapsulation. This restriction should be relaxed. In other words, it is to turn partial quantifications $\forall\underline{\partial}$ of derived dynamizers to fully quantified ones $\forall\partial$ if they are proven safe. For example, `double_dynamic` function should be able to have the following type scheme with a full quantification of the poly-dynamic variable ∂:

$$\texttt{double\_dynamic} : \forall\partial.\partial \rightarrow \texttt{dyn}$$

The application of this function yields a generalization check of its argument, just like the `dynamic` primitive. If the argument can be generalized, it is dynamized with a polymorphic type scheme. For example,

```
# double_dynamic (fun x → x);;
```

will create a dynamic value of $\forall\partial.(\partial \rightarrow \partial) * (\partial \rightarrow \partial)$.

However, not all the derived dynamizers can have full poly-dynamic quantifications. For example:

```
let apply_and_dynamize f =
  let v = f () in
  dynamic v
;;
```

Suppose we have an application `apply_and_dynamize (fun () → ref [])` and its argument is generalizable locally as $\forall\partial.\texttt{unit} \rightarrow \partial\ \texttt{list ref}$. What `apply_and_dynamize` actually dynamize is not the function itself, but the result of the application `(fun () → ref []) ()` $\rightsquigarrow$ `ref []`

whose type is ∂ `list ref`. Though the same ∂ type variable was regarded as encapsulable at the generalization of the function, it is now clearly ungeneralizable and non-encapsulable at `ref []`. Therefore we cannot give a full quantification to `apply_and_dynamize` and it must have only a partially quantified type scheme $\forall\underline{\partial}.(\texttt{unit} \to \partial) \to \texttt{dyn}$.

These examples indicate us that to allow derived dynamizers polymorphic encapsulation, we need a more complicated type system than before. The type generalization of derived dynamizers must investigate their definitions to discover whether we can safely give full poly-dynamic quantifications. This check will go as follows: once we find a let definition defines a derived dynamizer, we type check the definition again, temporarily quantify its arguments' poly-dynamic type variables. This generalization can be justified from the typing rules of dynamizers, that is, they must take arguments with closed type schemes. In this temporary typing, if an internal dynamizer is applied to an arguments with non-generalizable poly-dynamic variables, the corresponding poly-dynamic variables in the original typing cannot gain full-quantifications. Let's look a simple example:

```
let d = fun f g →
  let v = f () in
  let v' = g () in
  dynamic (v, g)
;;
```

The definition of `d` is typed as $(\texttt{unit} \to \partial_1) \to (\texttt{unit} \to \partial_2) \to \texttt{dyn}$. Before generalize it, we type the functional body again, but with the modified bindings $\{\texttt{f} : \forall\partial_1.\texttt{unit} \to \partial_1, \texttt{g} : \forall\partial_2.\texttt{unit} \to \partial_2\}$ in the typing environment. Thanks to these quantifications, `v` and `v'` have ∂'_1 and ∂'_2, different type variables from ∂_1 and ∂_2. Both types cannot be generalized since their definitions are expansive. At the call of the `dynamic` primitive, we find that `v` must not be polymorphically encapsulated but `g` can be. This indicates that in the original typing, ∂_2 can be full-quantified, but ∂_1 cannot: the type scheme of `d` is $\forall\underline{\partial_1}\partial_2.(\texttt{unit} \to \partial_1) \to (\texttt{unit} \to \partial_2) \to \texttt{dyn}$.

This full retyping of definitions of derived dynamizer will surely inflate the complexity of the typing. Further studies are required to find a simple solution, even if it might be less powerful than the full retyping.

4.7 Dynamics and constraint typing system

To distinguish the dynamic type variables related to dynamizations from the others, we introduced the poly-dynamic attribute ∂ to them. When the generic constraint type system will be introduced in Chapter 8, this poly-dynamic attribute can be easily transfered into generic constraints and we can make the type algebra simpler, using the same scheme proposed in Section 8.11.

This main idea of Section 8.11 is, roughly speaking, adding a new derived constraint $\texttt{generic}(\alpha)$ to the constraints of generic primitives instead of carrying the dynamic attribute of type schemes. For example, a generic primitive `print` of a constraint $[\forall\delta.\delta \to \texttt{unit}]$ will have the following new constraint:

$$\mathcal{K}_{\texttt{print}} \quad = \quad [\forall\alpha.\alpha \to \texttt{unit}\ \{\texttt{generic}(\alpha)\}]$$

By adding the constraint $\texttt{generic}(\alpha)$, the type variable α is regarded as generic and the constraint is distinguished from a constraint $[\forall\alpha.\alpha \rightarrow \texttt{unit}]$ for a non-generic value.

Like the attribute $\texttt{generic}(\alpha)$, we can transfer the poly-dynamic attribute of the type scheme of the `dynamic` primitive into its constraint as a new $\texttt{poly}(\alpha)$ constraint:

$$\mathcal{K}_{\texttt{dynamic}} \quad = \quad [\forall\alpha.\alpha \rightarrow \texttt{dyn}\ \{\texttt{poly}(\alpha)\}]$$

The poly-dynamic attributes of derived dynamizers can be deduced from their derived constraints on the `dynamic` primitive. For example, `double_dynamic` will have the following generic constraint:

$$\texttt{double\_dynamic : } [\forall\alpha.\alpha \rightarrow \texttt{dyn}\ \{\ \alpha * \alpha \rightarrow \texttt{dyn} <: \mathcal{K}_{\texttt{dynamic}}\}]$$

which corresponds with its current type scheme

$$\forall\underline{\partial}.\partial \rightarrow \texttt{dyn}$$

4.8 Conclusion

As an example of generic primitives, we have implemented the primitives of dynamic values over extensional polymorphism. Using the type dispatching mechanism of extensional polymorphism compilation, the compilations of the primitives are straightforwardly obtained as functions which take type information they need as run time type arguments. More than a mere example of generic primitives, dynamic values provide a simple run time type checking. The user-defined generic values we will introduce in Chapter 7 also perform type operations, but these operations will be limited to something simulatable and verifiable even at type checking phase. At this point, dynamic values are more dynamic than user-defined generics. Dynamic value framework can be used to overcome problems of user-defined generics which come from their staticness.

The latter half of this chapter, we argued polymorphic encapsulation of dynamic values. Since type variables inside dynamic values can be exported outside of their scope, they must be generalizable for type security. To encapsulate values with polymorphic types, we added a new typing of the `dynamic` primitive to check the generalization of its argument. Derived dynamizers, derived generic polymorphic functions which create dynamic values, are also easily defined in extensional polymorphism. We found, however, some of derived dynamizers cannot create polymorphic encapsulation, even when their arguments can be generalizable. Therefore we added another typing to inhibit all the derived dynamizer to create polymorphic encapsulation. This is a limitation of our system for simplicity. It is a future issue to give the polymorphic encapsulation capability also to derived dynamizers, if they are proven safe. We also needed a modification to the `coerce` primitive: to coerce back polymorphic contents of dynamic values to polymorphic, the coercion must be a special form of expression.

Dynamic values are exportation of types by encapsulating them inside of values. Therefore type information of dynamic values must be independent of any typing context. The type generalization check of the `dynamic` primitive assures the universality of dynamic type information inside *one* program. In following chapters (Chapter 5 and 6) we will consider universal data type identification throughout ML programs to build a type safe system of ML value exchange even between different programs.