Chapter 8

Constraint typing

8.1 Introduction

Detection of illegal generic instances

We have introduced generic bindings which permit us to define generic values by declaring list of type cases. The semantics of these "user-defined" generic values — selection of these type cases is by run time type matching, which can be naturally compiled to normal pattern match expressions. We also provided a simple extension of the GML type system for user generics. This simplicity of the typing system extension is achieved by ignoring an important property of user-defined generic values: even if a user-defined generic value has a type scheme σ in the type system, its semantics is not defined for all the instances of σ, in the sense that in some evaluation type context, it may not be evaluated to a semantic values, but *gerr*, the run time type error. This *non-exhaustiveness* of user generics cannot be taken into account, as far as we just give the anti-unifier of type patterns as the type scheme of a user-defined generic value.

Even though generic run time errors are reported as run time exceptions and they can be caught, it should be preferable if they can be detected at compile time statically. Otherwise, invalid use of generic instances must be found by trial and error basis debugging. Therefore in this chapter, we examine and challenge problems of the previous type system, and propose a solution, *generic constraint typing system* which can handle generic non-exhaustiveness.

Two-layer design

We are *not* going to develop this generic constraint typing system as an extension of the type system we have used in the previous chapters. Non-exhaustiveness information of generic values which will be called *generic constraints* cannot be expressed as type schemes. Moreover, they become quite complex: constraints of derived generic values must contain non-exhaustiveness information of other values, and constraints of recursively defined generic values are also recursive. Integrating such complex type information directly into the current type system will surely destroy its theoretical simplicity of ML typing.

Generic constraint typing system is defined as a new typing layer independent of the "normal" generic core type system. This two-layer design makes both type algebra simpler. The generic constraint typing verifies types of expressions which are supposed to be already type-checked in the normal typing layer. Under this assumption, generic constraint typing rules can focus just on generic related expression constructs. The other parts are granted whatever types they have. This separation also benefits implementation of extensional polymorphism. Type reconstruction has two phases corresponding to the two typing layers: the first reconstruction phase is a normal ML type reconstruction with extension of dynamic type variables. This *normal reconstruction phase* treats generic values as parametric polymorphic values. Then type-annotated expressions are passed to the second *generic type reconstruction phase*, which detects illegal uses of generic instances and tries to instantiate over-general types of generic instances to something valid. Thus the generic type reconstruction phase is independent from the normal reconstruction, therefore we can implement them separately. This is important when we try to implement extensional polymorphism as an extension of some existing ML implementation.

Outline of this chapter

We first points out the typing problem of core system with user-defined generic values in Section 8.2. Then in Section 8.3 we informally introduce the notion of full and polymorphic instances which distinguishes instances by their type annotation generalization. In Section 8.4 we define generic constraints, the type information of the generic constraint typing. Section 8.5, 8.6 and 8.7 define type instantiation $\tau <: \mathcal{K}$ of generic constraints to type expressions. Then we back to the formal full and polymorphic instances using generic constraints in Section 8.8. The type system itself is defined in Section 8.9, and 8.10 proves its soundness.

8.2 The limitation of the generic core typing system

As we have proven, the generic core type system including generic bindings for user generics is type-sound in some sense. That is, it makes sure that its well-typed programs are never evaluated into *err*: in other words, their compilation never fall into segmentation faults or bus errors when they are executed by an ML run time system. The other, generic run time error, *gerr* is raised when we have incorrect uses of user-defined generic values. If they are evaluated under types which do not match with any of their type patterns. For example:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
;;
val plus : ∀δ.δ → δ → δ
```

This `plus` generic function is defined only for the types `int` → `int` → `int` and `float` → `float` → `float`. Therefore, using `plus` in another type raises *gerr*:

```
# print_string (plus "hello" "world");;
Uncaught exception: Type_match_failure
```

The type pattern matching of user generics is compiled to IML's normal pattern matching. Therefore, as in the normal pattern matching, a run time type match failure *gerr* is reported as an exception. This exception can be easily handled by catching.

Normal ML typing does not check pattern match failures in a static type system but leave them to be found at run time. Just like pattern match failures, should we take the generic run time error for granted? Our answer here is no, from the following reasons.

First of all, in the GML semantics, the generic run time error is essentially an error of types, while normal pattern match failure is an error on values. The generic run time error is reported as a run time exception, but this is a property derived from our compilation method which translates user generics to normal pattern matching, and has no relationship with the semantics themselves. The above justification of generic run time errors as pattern match failure exception has no sense if we use different compilation scheme of user generics. Actually, in Chapter 9, we will introduce a new user generic value compilation without type pattern matching at run time.

In addition, since we have just finite number of type cases in generic bindings while the set of types is infinite, in almost cases user-defined generic values are non-exhaustive unless they have the default type cases. This means that user-defined generic values can be easily misused under evaluation type contexts undefined for them. Therefore rather than reported as run time exceptions, the incorrect generic instances should be detected in the type checking phase statically, otherwise run time type matching errors would suffer seriously programming productivity of extensional polymorphism.

This task – static detection of illegal generic instances – is impossible for the generic core type system, because of its ignorance of the type case information, i.e. non-exhaustiveness of generic values. To detect illegal generic instances, we must add *generic constraints*, non-exhaustive information of user generics to type systems.

8.3 Full and polymorphic instances

We first need to distinguish generic instances into two groups, instances whose validity can be checked at the point of their occurrences or not.

Full instances

We say an instance is a *full instance*, if its type annotation has no type variables generalized by some let or generic generalization. For example, in the following program,

```
generic plus = case
...
in
plus 1 2;;
```

the instance `plus` in the last line is *fully instantiated*. Evaluation type contexts of full instances are the same as their type annotations, because their annotations are not generalized. Therefore according to their annotations we can check whether they are evaluated correctly without any generic run time type error or not. Full instances which could fail into the generic run time error are regarded as *invalid* and rejected. Only the *valid* full instances which are proven to be safe are permitted in the constraint type system.

Polymorphic instances

On the other hand, instances whose types have dynamic type variables generalized somewhere outside are called *polymorphic instances*. For example, we have such polymorphic instances in definitions of derived polymorphic values:

```
let double x = plus x x in ...;;
```

Here the instance of `plus` is a polymorphic instance. The evaluation type context of `plus` is dynamic and unknown at the point of the definition of `double`, until full instantiations of `double`, like `double 1`. Validity of polymorphic instances is dependent on how the values which use them are fully instantiated. Therefore we delegate our validity investigation against polymorphic instances to checks of full instances of the values which use them.

The distinction of full and polymorphic instances are formally defined in Section 8.8.

8.4 Generic constraints

Now we introduce new type information for generic values, *generic constraints* (or just constraints in short), which indicate non-exhaustiveness information of generics. fully type instantiations of a generic value is restricted to those which satisfy these constraints.

8.4.1 Type patterns

For simplest cases, constraints for generic values are exactly the lists of type patterns. For example, the following simple `print_non_list` generic function

```
generic print_non_list = case
| string → unit ⇒ print_string
| int → unit ⇒ print_int
```

has the following constraint, which is identical to the type patterns of the definition:

$$\texttt{print\_non\_list} : [\ \texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\ ]$$

This form of constraint is called *type patterns* too. If `print_non_list` is fully instantiated, its type must be an instance of one of the type patterns, therefore, `string` → `unit` or `int` → `unit`. These two full instance types are called *(valid) instances* of the constraint $[\,\texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\,]$. We use the symbol <: to denote this relationship. Here, for example,

$$\texttt{int} \rightarrow \texttt{unit} \;<:\; [\,\texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\,]$$

denotes that $\texttt{int} \rightarrow \texttt{unit}$ is one of the valid instances of the constraint.

8.4.2 Derived constraints

Thus constraints indicate in which types generic values can be instantiated without any generic run time error in evaluation. But things become bit more complicated if a `generic` binding uses other generic values inside:

```
generic print_list_also = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print_non_list l
| ∀δ.δ → unit ⇒ print_non_list
```

In this case, the constraint of `print_list_also` is not just

$$[\,\forall\delta.\delta\;\texttt{list} \rightarrow \texttt{unit} \mid \forall\delta.\delta \rightarrow \texttt{unit}\,]$$

We cannot use this generic value for all the instances of the type patterns, because the both type method expressions contain polymorphic instances of `print_non_list`. For example, suppose that `print_list_also` is instantiated as $\tau = S(\delta\;\texttt{list} \rightarrow \texttt{unit})$ for some substitution S, using the first type case. Then in order to evaluate this instance without any run time type error, the occurrence of `print_non_list` in the type method expression with the type $\delta \rightarrow \texttt{unit}$ must be also a valid full instance under S: $S(\delta \rightarrow \texttt{unit})$ has to be a valid full instance type of the constraint of `print_non_list`, $[\,\texttt{int} \rightarrow \texttt{unit} \mid \texttt{float} \rightarrow \texttt{unit}\,]$. To express this further constraints derived from polymorphic instances of non-exhaustive generic values, we introduce *derived constraints* $\tau <: \mathcal{K}$, where $\mathcal{K}$ is a constraint of such a polymorphic instance. Now a type pattern in constraints accompanies with a (possibly null) list of derived constraints $\{\tau_1 <: \mathcal{K}_1 \;\&\; \ldots \;\&\; \tau_n <: \mathcal{K}_n\}$. `print_list_also` must have a generic constraint with these derived constraints:

$$\begin{aligned}\texttt{print\_list\_also}: \;& [\,\forall\delta.\delta\;\texttt{list} \rightarrow \texttt{unit}\;\{\delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\}\\ & \mid \forall\delta.\delta \rightarrow \texttt{unit}\;\{\delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\}\,]\end{aligned}$$

Here $\mathcal{K}_{\texttt{print\_non\_list}} = [\,\texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\,]$. The instantiation of `print_list_also` is now restricted to $\texttt{string list} \rightarrow \texttt{unit}$, $\texttt{int list} \rightarrow \texttt{unit}$, $\texttt{string} \rightarrow \texttt{unit}$ and $\texttt{int} \rightarrow \texttt{unit}$, with these derived constraints.

Derived constraints are also used for constraints of let bound derived generic polymorphic values:

$$\begin{aligned}&\texttt{let print\_tuple (x,y) = print\_non\_list x; print\_non\_list y;;}\\ &\texttt{print\_tuple}: \;[\,\forall\delta\delta'.\delta * \delta' \rightarrow \texttt{unit}\;\{\,\delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}} \;\&\\ &\qquad\qquad\qquad\qquad\qquad\qquad \delta' \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\}\,]\end{aligned}$$

Because there is no type matching in `print_tuple`, its constraint consists of only one element of type pattern constraint with derived constraints. The type safety of `print_tuple` depends on those of two polymorphic instances of `print_non_list` inside the definition.

8.4.3 Recursive constraints

We also need some recursive constructions of constraints for those of recursive generic values. We introduce *constraint variables* κ and *the fixed point operator* μ to express the *recursive constraints*. For example, `print_rec`, a revised version of `print_list_also` which can print integer or string lists of any depth:

```
generic rec print_rec = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print_rec l
| ∀δ.δ → unit ⇒ print_non_list
```

has a constraint $\mathcal{K}_{\texttt{print\_rec}}$ where,

$$\begin{aligned}\mathcal{K}_{\texttt{print\_rec}} = \mu\kappa.\,[\ & \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \kappa\ \}\\ |\ & \forall\delta.\delta \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\ \}]\end{aligned}$$

If a type τ is a valid full instance of this constraint, then τ must satisfy one of the followings:

- $\tau \leq_S \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}$ and $S(\delta \rightarrow \texttt{unit}) <: \mathcal{K}_{\texttt{print\_rec}}$
- or, $\tau \leq_S \forall\delta.\delta \rightarrow \texttt{unit}$ and $S(\delta \rightarrow \texttt{unit}) <: \mathcal{K}_{\texttt{print\_non\_list}}$

The recursive constraint is *μ-expanded* (defined formally later) and the constraint variable κ is substituted by the constraint itself.

8.4.4 Constraints must be isomorphic to the structure of the type cases

Since `print_list_also` can be instantiated to only four types `string list` $\rightarrow$ `unit`, `string` $\rightarrow$ `unit` and `int` $\rightarrow$ `unit`, someone may think that he could give a much simpler constraint for `print_list_also`:

$$\begin{aligned}\texttt{print\_list\_also} :\ [\ & \texttt{string list} \rightarrow \texttt{unit}\\ |\ & \texttt{int list} \rightarrow \texttt{unit}\\ |\ & \texttt{string} \rightarrow \texttt{unit}\\ |\ & \texttt{int} \rightarrow \texttt{unit}\ ]\end{aligned}$$

Clearly, if a type τ is a valid full instance of the original constraint, then it is also a valid full instance of the second one and vice versa; both constraints express the same set of possible full instances, therefore they could be equivalent. This task – to have "simpler" equivalent constraints — would seem to be easy if we just look this non-recursive example, but once we introduce recursive constraints, the problem becomes much harder. Defining the clear "normal forms of constraints" seems to be almost impossible, due to the existence of recursive constraints. Therefore we do not try to introduce such equivalence between constraints. Keeping constraints as they are, isomorphic to generic value definitions also has a big advantage. We can optimize the type case selection greatly by pre-computing all the type matchings at the compilation phase, because constraints have the complete information how the type matchings will happen. The run time type matching of the IML compilation can be simplified just into accessing arrays which contain each type method expression. We will discuss this *generic flow* optimization later in Chapter 9.

8.4.5 Formal definition

Now we give the formal definition of *generic constraints*:

$$
\begin{array}{rcll}
\mathcal{K} & ::= & [k|\ldots|k] & \text{Type pattern lists} \\
& | & \kappa & \text{Constraint variables} \\
& | & \mu\kappa.[k|\ldots|k] & \text{Recursive constraints} \\
\\
k & ::= & \sigma\ D & \text{Type patterns} \\
\\
D & ::= & \{d\ \&\ \ldots\ \&\ d\} & \text{Derived constraint lists} \\
\\
d & ::= & \tau <: \mathcal{K} & \text{Derived constraints}
\end{array}
$$

A type pattern constraint list $[k_1|\ldots|k_n]$ means that it can be fully instantiated to a type which is one of the instances of the type patterns k_i. A type pattern takes a (possibly null) list of *derived constraints* $D = \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \tau_n <: \mathcal{K}_n\}$ which expresses further derived instantiation conditions. The null derived constraint list $\{\,\}$ is usually omitted to write. For example, a constraint $[\sigma\ \{\,\}]$ is written just $[\sigma]$ in short, because it can be fully instantiated to any instance of σ. For example, a non-generic value whose type scheme is σ in the normal typing layer will have a constraint $[\sigma]$, a singleton type pattern list constraint of the type scheme. The constraint $[\sigma]$ is often written simply as σ.

We also have μ *operator* and *constraint variables* κ to express recursively defined constraints. We use the symbol $\mathbb{K}$ for the set of the constraint variables. When a constraint is μ-abstracted, it must be a type pattern constraint $[k_1|\ldots|k_n]$. Type pattern lists $[k_1|\ldots|k_n]$ and derived constraint lists $\{d_1\ \&\ \ldots\ \&\ d_n\}$ are often written as $[k_i]_{i=1}^n$ and $\{d_i\}_{i=1}^n$ respectively in short. Sometimes index ranges of these lists are also omitted as $[k_i]$ and $\{d_i\}$, if it is clear from the context.

A constraint is called *non-generic* when it is a singleton type pattern list $[\sigma\ \{\}]$ with an empty derived constraint list where σ has no bound dynamic type variable. Otherwise, it is called *generic*. The formal definition of this predicate *generic* is given as follows:

$$
\begin{array}{l}
\mathit{generic}([\sigma\ \{\}]) = \mathit{false} \quad \text{when } BDV(\sigma) = \emptyset \\
\text{Otherwise, } \mathit{generic}(\mathcal{K}) = \mathit{true}
\end{array}
$$

Coercion of constraints to type schemes

In Chapter 7, type schemes of user-defined generic values are defined as the least general anti-unifier of the type patterns. On the other hand, generic constraints of generic values have isomorphic structure to their type patterns. Therefore, generic constraints can be regarded as extended type schemes and a downward coercion from constraints to type schemes can be defined as follows:

$$\begin{array}{rcl} \mathit{coerce}([\sigma\ D]) & = & \sigma \\ \mathit{coerce}([\sigma_i\ D_i]_{i=1}^{n>1}) & = & \mathit{p\_antiunif}(\{\sigma_i\}_{i=1}^{n}) \\ \mathit{coerce}(\mu\kappa.\mathcal{K}) & = & \mathit{coerce}(\mathcal{K}) \end{array}$$

If a constraint is a singleton type pattern list, its coercion to a normal type scheme is the type scheme part of the list. Otherwise, a coercion of a constraint is defined as the type pattern unification of type schemes of its patterns. Actually, if we fix the constraint of a value in the constraint typing to $\mathcal{K}$, its type scheme in the normal typing is uniquely determined as $\mathit{coerce}(\mathcal{K})$.

Coercion of typing environment is also definable. Let Γ be a constraint typing environment from identifiers to constraints, then its coercion to a normal typing environment $\mathit{coerce}(\Gamma)$ is given as follows:

$$\mathit{coerce}(\Gamma) = \{(\mathtt{x} \Rightarrow \mathit{cnst}(\mathcal{K})) \mid (\mathtt{x} \Rightarrow \mathcal{K}) \in \Gamma\}$$

8.4.6 Type substitution

The notion of free type variables and type substitutions are naturally extended for constraints. In a type pattern constraint $\sigma\ D$, the free type variables in a derived constraint list D are bound by the quantified variables of σ:

$$\begin{array}{rcl} FV([k_i]_{i=1}^{n}) & = & \bigcup_{i=1}^{n} FV(k_i) \\ FV(\kappa) & = & \emptyset \\ FV(\mu\kappa.\mathcal{K}) & = & FV(\mathcal{K}) \\ FV(\forall V.\tau\ \{d_i\}_{i=1}^{n}) & = & FV(\forall V.\tau) \cup (\bigcup_{i=1}^{n} FV(d_i) \setminus V) \\ FV(\tau <: \mathcal{K}) & = & FV(\tau) \cup FV(\mathcal{K}) \end{array}$$

When we apply a substitution to a constraint $\mathcal{K}$n, we may need to rename the type variables in $\mathcal{K}$ which are bound by quantifications:

$$\begin{array}{rcll} S([k_i]_{i=1}^{n}) & = & [S(k_i)]_{i=1}^{n} & \\ S(\kappa) & = & \kappa & \\ S(\mu\kappa.\mathcal{K}) & = & \mu\kappa.S(\mathcal{K}) & \\ S(\sigma\ \{d_i\}_{i=1}^{n}) & = & S(\sigma)\ \{S(d_i)\}_{i=1}^{n} & \text{rename bound variables if needed} \\ S(\tau <: \mathcal{K}) & = & S(\tau) <: S(\mathcal{K}) & \end{array}$$

8.4.7 Constraint substitutions and μ-expansion

Free and bound constraint variable notions for μ-abstractions can be defined like the normal lambda terms. A *closed constraint* is a constraint which has no free constraint variable. More precisely, the definition of the function KFV to get the free constraint variables of a constraint is:

$$
\begin{array}{rcl}
KFV([k_i]_{i=1}^n) & = & \bigcup_{i=1}^n KFV(k_i) \\
KFV(\kappa) & = & \{\kappa\} \\
KFV(\mu\kappa.\mathcal{K}) & = & KFV(\mathcal{K}) \setminus \{\kappa\} \\
KFV(\sigma\ \{d_i\}_{i=1}^n) & = & \bigcup_{i=1}^n KFV(d_i) \\
KFV(\tau <: \mathcal{K}) & = & KFV(\mathcal{K})
\end{array}
$$

For expansions of the fixed point operator, the μ-abstraction, we need to define *constraint substitutions*. Constraint substitutions are substitutions from constraint variables to constraints. We use ψ meta-variables for these substitutions. The definition of constraint substitutions as functions from a constraint to a constraint is as follows. Like the normal substitutions for type expressions and schemes, bound constraint variables are renamed if necessary:

$$
\psi \quad ::= \quad \{\kappa \Rightarrow \mathcal{K}, \ldots, \kappa \Rightarrow \mathcal{K}\}
$$

$$
\begin{array}{rcll}
\psi([k_i]_{i=1}^n) & = & [\psi(k_i)]_{i=1}^n & \\
\psi(\mu\kappa.\mathcal{K}) & = & \mu\kappa.\psi(\mathcal{K}) & \text{rename the bound variable if necessary} \\
\psi(\sigma\ \{d_i\}_{i=1}^n) & = & \sigma\ \{\psi(d_i)\}_{i=1}^n & \\
\psi(\tau <: \mathcal{K}) & = & \tau <: \psi(\mathcal{K}) &
\end{array}
$$

The *μ-expansion* of a recursive constraint can be defined by using constraint substitutions:

Definition 15 **(μ-expansion of recursive constraint)** *The* μ-expansion *of a recursive constraint $\mu\kappa.\mathcal{K}$ is a constraint $\{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(\mathcal{K})$. If $\mathcal{K}'$ is a μ-expansion of $\mu\kappa.\mathcal{K}$, we often write $\mu\kappa.\mathcal{K} \overset{\mu}{\to} \mathcal{K}'$.*

If a μ-abstracted constraint variable κ does not occur in the body of a recursive constraint $\mu\kappa.\mathcal{K}$, $\mu\kappa.\mathcal{K} \overset{\mu}{\to} \mathcal{K}$. This μ-abstraction is useless, therefore we regard the abstracted constraint equivalent to its body. This is the only structural equivalence we introduce in constraints except the trivial α-conversions of quantified type variables and constraint variables:

Definition 16 **(Constraint equivalence)** *Let $\mathcal{K}$ be a constraint and κ be a constraint variable such that $\kappa \notin KFV(\mathcal{K})$. Then we say the recursive constraint $\mu\kappa.\mathcal{K}$ is equivalent to $\mathcal{K}$, and write $\mu\kappa.\mathcal{K} \equiv \mathcal{K}$.*

For example, $\mu\kappa.[\sigma\ \{\}] \equiv [\sigma\ \{\}]$. This equivalence is trivially generalized for constraints:

$$
\begin{array}{rcll}
\mu\kappa.\mathcal{K} & \equiv & \mathcal{K} & \text{when } \kappa \notin KFV(\mathcal{K}) \\
[k_i]_{i=1}^n & \equiv & [k'_i]_{i=1}^n & \text{when } k_i \equiv k'_i \\
\sigma\ \{d_i\}_{i=1}^n & \equiv & \sigma\ \{d'_i\}_{i=1}^n & \text{when } d_i \equiv d'_i \\
\tau <: \mathcal{K} & \equiv & \tau <: \mathcal{K}' & \text{when } \mathcal{K} \equiv \mathcal{K}'
\end{array}
$$

8.5 Valid full instances

In the generic constraint typing, the full instances of generic values are "constrained" by their constraints. In this section, we define the first version of the set of *valid full instance types* of constraints. This is called *strict* valid full instance relationship, since it is too restricted and not practical actually. But because of the simplicity of its definition, we start from this strict predicate.

Unfortunately, due to the existence of recursive constraints, the strict valid full instance predicate is undecidable. Therefore, in Section 8.6, we introduce a restriction to the *admissible* set of generic constraints in the generic constraint typing, so that the predicate can be decidable within the admissible constraints.

Strict valid full instance relationship is too strong, since it assures not only error free evaluation of generic instances, but also termination of recursion on types. The latter is not required for the type safety of generic instances, and unnecessarily rejects generic instances which never terminates but never raises the generic run time error in evaluation. Later in Section 8.7, we remove this superfluous condition from the valid full instance relationship and obtain a *relaxed* version.

8.5.1 Full instance relationship

The instances are called *strictly valid full instances* or strict full instances. The word "strict" means that this relationship is bit too tight in practice. Later we will define another rather "relaxed" relationship called *relaxed full instances* where more types can be valid full instances of constraints, but at first, we would like to start from something simpler.

The strict full instance relationships $\tau \ll: \mathcal{K}$ between a type τ and a constraint $\mathcal{K}$ is defined by the following rules:

$$
\textbf{(k-case)}\ \frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \\ \tau \leq \sigma_p \quad \tau \ll: \sigma_p\ D_p \quad p \leq n\end{array}}{\tau \ll: [\sigma_i\ D_i]_{i=1}^n}
$$

$$
\textbf{(k-rec)}\ \frac{\tau \ll: \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}}{\tau \ll: \mu\kappa.\mathcal{K}}
$$

$$
\textbf{(k-derive)}\ \frac{\tau \leq_S \sigma, \quad \forall 1 \leq i \leq n \ \ S(\tau_i) \ll: S(\mathcal{K}_i)}{\tau \ll: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n}
$$

In (k-case), we always select the first pattern $\sigma_p\ D_p$ where $\tau \leq \sigma_p$. The other latter $\sigma_i\ D_i$'s are ignored, even if $\tau \leq \sigma_i$. This corresponds with the pattern matching compilation of generic type case selection (Chapter 3): if there are more than one matches, the first one is always used.

The following algorithm *kinst* is to check whether a given type is a full instance of a constraint or not. For a technical reason for future proofs, μ-expansions are applied lazily. Lazy applications of μ-expansions are stored inside the constraint substitution ψ of the first argument of *kinst*:

$$\begin{array}{rcl}
\mathit{kinst}(\psi, \tau \ll: [\sigma_i\ D_i]_{i=1}^n) & = & \exists p, \text{where } \tau \not\leq \sigma_1\ \&\ \ldots \tau \not\leq \sigma_{p-1}\ \&\ \tau \leq \sigma_p \\
& & \&\ \mathit{kinst}(\psi, \tau \ll: \sigma_p\ D_p) \\
\mathit{kinst}(\psi, \tau \ll: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n) & = & \tau \leq_S \sigma\ \&\ \forall_{i=1}^n \mathit{kinst}(\psi, S(\tau_i) \ll: S(\mathcal{K}_i)) \\
\mathit{kinst}(\psi, \tau \ll: \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
& & \mathit{kinst}(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \tau \ll: \mathcal{K}) \\
\mathit{kinst}(\psi, \tau \ll: \kappa) & = & \text{if } \kappa \notin \mathit{Dom}(\psi) \text{ then } \mathit{false} \text{ else } \mathit{kinst}(\psi, \tau \ll: \psi(\kappa))
\end{array}$$

Theorem 25 (Correctness of *kinst* against $\ll:$) *Let τ be a type, $\mathcal{K}$ be a constraint, and ψ be a constraint substitution contains μ-expansions such that $\psi \circ \psi = \psi$. If and only if $\mathit{kinst}(\psi, \tau \ll: \mathcal{K})$ terminates and returns true, we have $\tau \ll: \psi(\mathcal{K})$.*

PROOF: First we prove $\mathit{kinst}(\psi, \tau \ll: \mathcal{K}) = \mathit{true} \implies \tau \ll: \psi(\mathcal{K})$ by induction over the *kinq* algorithm:

Type pattern constraints Let $\mathcal{K} = [\sigma_i\ D_i]$. Since $\mathit{kinst}(\psi, \tau \ll: \mathcal{K}) = \mathit{true}$, there must be an integer p such that $\tau \not\leq \sigma_1, \ldots, \tau \not\leq \sigma_{p-1}, \tau \leq \sigma_p$ and $\mathit{kinst}(\psi, \tau \ll: \sigma_p\ D_p) = \mathit{true}$. Let $D_p = \{\tau_{pi} <: \mathcal{K}_{pi}\}$ and S be an instantiation such that $\tau \leq_S \sigma_p$, then for all i, $\mathit{kinst}(\psi, S(\tau_{pi}) \ll: S(\mathcal{K}_{pi})) = \mathit{true}$. By induction hypothesis, $S(\tau_{pi}) \ll: \psi(S(\mathcal{K}_{pi}))$. Since $\psi(S(\mathcal{K}_{pi})) = S(\psi(\mathcal{K}_{pi}))$, using (k-derive) we obtain $\tau \ll: \sigma_p\ \{\tau_{pi} <: \psi(\mathcal{K}_{pi})\}$ and by (k-case), it leads to $\tau \ll: [\sigma_i\ \psi(D_i)]$, which is equal to $\tau \ll: \psi(\mathcal{K})$.

Constraint variable Let $\mathcal{K} = \kappa$. Since $\mathit{kinst}(\psi, \tau \ll: \kappa) = \mathit{true}$, ψ must have a binding for κ, and $\mathit{kinst}(\psi, \tau \ll: \psi(\kappa)) = \mathit{true}$. By induction hypothesis, $\tau \ll: (\psi \circ \psi)(\kappa)$. From the idempotency of ψ, this is equal to $\tau \ll: \psi(\kappa)$.

Recursive constraints Let $\mathcal{K} = \mu\kappa'.\mathcal{K}'$. From $\mathit{kinst}(\psi, \tau \ll: \mu\kappa.\mathcal{K}') = \mathit{true}$, If we let $\psi' = \psi \setminus \{\kappa' \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\}$,

$$\mathit{kinst}(\psi'', \tau \ll: \mathcal{K}') = \mathit{true}$$

From the idempotency of ψ, ψ' is also idempotent and ψ'', too. Therefore we can use the induction hypothesis and obtain $\tau \ll: \psi''(\mathcal{K}')$.

Since $\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\} = \{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\} \circ \psi'$ from its idempotency, $\tau \ll: \{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\}(\psi'(\mathcal{K}'))$. Using (k-rec) we obtain $\tau \ll: \mu\kappa'.\psi'(\mathcal{K}')$, where $\mu\kappa'.\psi'(\mathcal{K}') = \psi(\mu\kappa'.\mathcal{K}')$.

Second, we prove the converse, $\tau \lll: \psi(\mathcal{K}) \Longrightarrow kinst(\psi, \tau \lll: \mathcal{K}) = true$ by induction over the derivation of the predicate:

(k-case) It is the case when $\psi(\mathcal{K}) = [\sigma_i \ D_i]$. The last step of its proof is as follows:

$$\frac{\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \tau \leq \sigma_p \quad p \leq n \quad \tau \lll: \sigma_p \ D_p}{\tau \lll: [\sigma_i \ D_i]_{i=1}^n}$$

If we let $D_p = \{\tau_j <: \mathcal{K}_j\}_{j=1}^m$, The predicate $\tau \lll: \sigma_p \ D_p$ must be proven by (k-derive) as follows:

$$\frac{\tau \leq_S \sigma_p, \quad \forall 1 \leq j \leq m \ \ S(\tau_j) \lll: S(\mathcal{K}_j)}{\tau \lll: \sigma \ \{\tau_j <: \mathcal{K}_j\}_{j=1}^m}$$

The constraints $\mathcal{K}_j$ are not be affected by ψ, since they are sub-constraints of $\psi(\mathcal{K})$ and ψ is idempotent. Therefore the predicate $S(\tau_j) \lll: S(\mathcal{K}_j)$ can be rewritten to $S(\tau_j) \lll: \psi(S(\mathcal{K}_j))$. Applying the induction hypothesis to these predicates, we have $kinst(\psi, S(\tau_j) \lll: S(\mathcal{K}_j)) = true$. They leads to $kinst(\psi, \tau \lll: \sigma_p \ D_p) = true$. From it and $\tau \not\leq \sigma_1, \ \ldots, \ \tau \not\leq \sigma_{p-1}$, we obtain $kinst(\psi, \tau \lll: \mathcal{K})$.

(k-rec) It is when $\psi(\mathcal{K}) = \mu\kappa'.\mathcal{K}'$. The proof ends as follows:

$$\frac{\tau \lll: \{\kappa' \Rightarrow \mu\kappa'.\mathcal{K}'\}\mathcal{K}'}{\tau \lll: \mu\kappa'.\mathcal{K}'}$$

When $\mathcal{K}$ is not a constraint variable

If $\mathcal{K}$ is not a constraint variable, it must be a μ-abstraction and can be written as $\mathcal{K} = \mu\kappa'.\mathcal{K}''$, where $\psi(\mathcal{K}'') = \mathcal{K}'$. The premise is rewritten as

$$\tau \lll: \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}(\psi(\mathcal{K}''))$$

Since we can suppose κ' is out of ψ, $\{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\} = \{\kappa' \Rightarrow \psi(\mu\kappa'.\mathcal{K}'')\}$ and

$$\{\kappa' \Rightarrow \psi(\mu\kappa'.\mathcal{K}'')\} \circ \psi = \psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}$$

Therefore $\tau \lll: (\psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\})\mathcal{K}''$. From this and using the induction hypothesis, we have $kinst(\psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}, \tau \lll: \mathcal{K}'') = true$. This is equal to

$$kinst(\psi, \tau \lll: \mu\kappa'.\mathcal{K}'') = true$$

When $\mathcal{K}$ is a constraint variable

If $\mathcal{K}$ is a constraint variable κ, $\psi(\kappa) = \mu\kappa'.\mathcal{K}'$. Since ψ is a constraint substitution made of μ-expansions, $\kappa = \kappa'$. The premise is rewritten to

$$\tau \lll: \psi(\mathcal{K}')$$

Applying the induction hypothesis to the premise, we have: $kinst(\psi, \tau <: \mathcal{K}') = true$.

On the other hand,

$$\begin{aligned} kinst(\psi, \tau <: \kappa) &= kinst(\psi, \tau <: \psi(\kappa)) \\ &= kinst(\psi, \tau <: \mu\kappa'.\mathcal{K}') \\ &= kinst(\psi'', \tau <: \mathcal{K}') \end{aligned}$$

where $\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\}$ and $\psi' = \psi \setminus \{\kappa' \Rightarrow \psi(\kappa')\}$. From the idempotency of ψ and $\psi' = \psi \setminus \{\kappa' \Rightarrow \mu\kappa'.\mathcal{K}'\}$, $\psi'' = \psi$. Therefore $kinst(\psi, \tau <: \kappa) = kinst(\psi, \tau <: \mathcal{K}') = true$.

□

Unfortunately this $\ll:$ and *kinst* algorithm have a serious problem. Because of (k-rec) recursive rule, reconstruction of derivations for recursive constraints sometimes never terminates. For example, let $\mathcal{K} = \mu\kappa.[\forall\delta.\delta\ \{\delta * \delta <: \kappa\}]$. Then $kinst(\emptyset, \tau \ll: \mathcal{K})$ never terminates, since it recursively calls $kinst(\psi, \tau * \tau \ll: \mathcal{K})$, $kinst(\psi', (\tau * \tau) * (\tau * \tau) \ll: \mathcal{K})$,... infinitely. More generally, this full instance relationship $\ll:$ is *undecidable* [DRW95]. Therefore we need to restrict the generic constraints so that the relationship can be decidable in the restricted set of constraints.

8.6 Admissible constraints

We define *the size of a type*, an arithmetic expression which denotes complexity of a type and restrict the set of constraints to *admissible* ones so that in the recursive checks of *kinst* for an admissible constraint, type sizes of the arguments can decrease and therefore terminate.

8.6.1 Type sizes

The type size of τ, $|\tau|$ is defined as a simple formal arithmetic expression as follows:

$$\begin{aligned} |base| &= 1 \\ |\alpha| &= 1 \\ |\delta_i| &= x_i \\ |\tau_1 \to \tau_2| &= \textstyle\sum_{i=1}^{2} |\tau_i| + 1 \\ |\tau_1 * \tau_2| &= \textstyle\sum_{i=1}^{2} |\tau_i| + 1 \end{aligned}$$

For each dynamic type variable δ_i, $|\delta_i|$ returns a variable x_i, specific to the type variable. It is clear that for any type τ, $|\tau|$ can be uniquely represented in the form of $\sum_i a_i x_i + c$, where $a_i > 0$ and $c \geq 0$.

The *equality* of type sizes is the arithmetic expression equality. We write $|\tau| = |\tau'|$ if both the expressions are same arithmetically. The *type size ordering* $<$ is defined as their ordering as equations by ranging all the variables x_i appears in the equations more than 0:

$|\tau| < |\tau'| \iff$ for all $x_i > 0$, $|\tau| < |\tau'|$ as integers

Checking $|\tau| < |\tau'|$ is done by proving this inequality by the following: Let $|\tau| = \sum_i a_i x_i + c$ and $|\tau'| = \sum_i a'_i x_i + c'$, supposing if x_i does not appear in $|\tau|$ (*resp.* $|\tau'|$) then $a_i = 0$ (*resp.* $a'_i = 0$). Then what we have to prove is:

$$\forall x_i > 0, \quad |\tau'| - |\tau| = \sum_i (a'_i - a_i)x_i + c' - c > 0,$$

and it is checked easily as follows:

- If there exists i such that $a'_i - a_i < 0$ then the inequality is not verified.
- Otherwise, the inequality is equivalent to $\sum_i (a'_i - a_i) + c' - c > 0$, which we can easily verify.

Example

$|\delta_1 * \delta_2| = x_1 + x_2 + 1$ and $|\delta_1 \to \delta_1| = x_1 + x_2 + 1$. Therefore

$$|\delta_1 * \delta_2| = |\delta_1 \to \delta_1|$$

$|\delta_1 \to \texttt{unit}| = x_1 + 2$ and $|\delta_1 * \delta_1 \to \texttt{unit}| = 2x_1 + 3$. Therefore

$$|\delta_1 \to \texttt{unit}| < |\delta_1 * \delta_1 \to \texttt{unit}|$$

□

Lemma 21 (Dynamic substitution preserves type size ordering) *Type substitutions whose domains are restricted within the dynamic type variables preserve the ordering of the type expressions: for a substitution S such that $Dom(S) \subset \mathbb{D}$, if $|\tau| < |\tau'|$ then $|S(\tau)| < |S(\tau')|$.*

Proof: Suppose $|\tau| < |\tau'|$ and let $|\tau| = \sum_i a_i x_i + c$ and $|\tau'| = \sum_i a'_i x_i + c'$ where $a_i \geq 0$ and $a'_i \geq 0$ and sharing the same set of x_i. Then $|S(\tau)|$ and $|S(\tau')|$ can be obtained by easy arithmetic manipulation:

$$|S(\tau)| = \sum_i a_i |S(\delta_i)| + c, \quad |S(\tau')| = \sum_i a'_i |S(\delta_i)| + c'$$

Since for all the variables $x'_i > 0$ in $|S(\tau)|$ and $|S(\tau')|$, $|S(\delta_i)| > 0$, the inequation $|S(\tau)| < |S(\tau')|$ is equivalent to $|\tau| < |\tau'|$ when $x_i = |S(\delta_i)|$. which is one of the cases of $\forall x_i > 0, |\tau| < |\tau'|$. □

Type size of substitution $|S|$

The type size of a substitution S, $|S|$ is defined using the type size $|\tau|$:

$$|S| = \sum_{\delta \in Dom(S)} |S(\delta)|$$

Type size of type instantiation $|\tau \leq \sigma|$

To make the discussion smoothly, we also need to define the *size of a predicate* $\tau \leq \sigma$. The size of a predicate is equivalent to the size of the type instantiation:

$$|\tau \leq \sigma| = |S| \quad \text{where } \tau \leq_S \sigma$$

We here suppose $Dom(S) = BV(\sigma)$ to have a unique size. Of course to be well defined, τ must be an instance of σ.

From the view of the compilation of extensional polymorphism, the type size of instantiation $|\tau \leq \sigma|$ indicates the type size of run time type dispatching for a generic full instance $\mathtt{x}_\sigma : \tau$:

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_\emptyset = \mathtt{x} \ll S(\delta_1) \gg \ \ldots \ \ll S(\delta_n) \gg \quad \text{where } \forall V.\tau' = \sigma \text{ and } [\delta_i] = \mathit{Vlist}(V \cap \mathbb{D}, \tau')$$

$|\tau \leq \sigma|$ is equal to the sum of the type sizes of $S(\delta_i)$.

Lemma 22 (Substitution preserves the order of $|\tau \leq \sigma|$) *Let τ and τ' be types, and σ and σ' be type schemes. If $|\tau \leq \sigma| < |\tau' \leq \sigma'|$, then for any substitution S_0, $|S_0(\tau) <: S_0(\sigma)| < |S_0(\tau') <: S_0(\sigma')|$.*

Proof: From the well-definedness of $|\tau \leq \sigma|$ and $|\tau' \leq \sigma'|$, there are substitutions S and S' such that $\tau \leq_S \sigma$, $\tau' \leq_{S'} \sigma'$ and:

$$\sum_{\delta \in BDV(\sigma)} |S(\delta)| \quad < \quad \sum_{\delta' \in BDV(\sigma')} |S'(\delta')|$$

From Lemma 4, $S_0(\tau) \leq S_0(\sigma)$, $S_0(\tau') \leq S_0(\sigma')$ and:

$$\begin{array}{rcccc} |S_0(\tau) <: S_0(\sigma)| & = & \sum_{\delta \in BDV(S_0(\sigma))} |(S_0 \circ S)(\delta)| & = & \sum_{\delta \in BDV(\sigma)} |(S_0 \circ S)(\delta)| \\ |S_0(\tau') <: S_0(\sigma')| & = & \sum_{\delta' \in BDV(S_0(\sigma'))} |(S_0 \circ S')(\delta')| & = & \sum_{\delta' \in BDV(\sigma')} |(S_0 \circ S')(\delta')| \end{array}$$

Because S is an admissible substitution, for any dynamic type variable δ, $FV(S(\delta)) \in \mathbb{D}$. Therefore if we let $S'_0 = \{(\delta \Rightarrow \tau) \in S_0\}$, a substitution same as S_0 whose domain is restricted only to the dynamic type variables, $|(S_0 \circ S)(\delta)| = |(S'_0 \circ S)(\delta)|$. By using S'_0, now the sizes are:

$$\begin{array}{rcl} |S_0(\tau) <: S_0(\sigma)| & = & \sum_{\delta \in BDV(\sigma)} |(S'_0 \circ S)(\delta)| \\ |S_0(\tau') <: S_0(\sigma')| & = & \sum_{\delta' \in BDV(\sigma')} |(S'_0 \circ S')(\delta')| \end{array}$$

Because $Dom(S'_0) \subset \mathbb{D}$, from Lemma 21 and the linearity of type sizes,

$$\sum_{\delta \in BDV(\sigma)} |(S'_0 \circ S)(\delta)| \quad < \quad \sum_{\delta' \in BDV(\sigma')} |(S'_0 \circ S)(\delta')|$$

and it is equal to $|S_0(\tau) \leq S_0(\sigma)| < |S_0(\tau') \leq S_0(\sigma')|$ □

Lemma 23 *Let σ_1 and σ_2 be type schemes and τ be a type expression such that $\tau \leq \sigma_1$ and $\tau \leq \sigma_2$. If σ_1 is more general than σ_2, i.e. $inst(\sigma_2) \leq \sigma_1$, $|\tau \leq \sigma_2| \leq |\tau \leq \sigma_1|$.*

PROOF: Let $\tau \leq_{S_1} \sigma_1$ and $\tau \leq_{S_2} \sigma_2$. Let $\sigma_2 = \forall V_2.\tau_2$ and $\tau_2 \leq_S \sigma_1$. Then $S_1 = S_2 \circ S$. $|\tau \leq \sigma_1| = \sum_{\delta \in BDV(\sigma_1)} |S_2(S(\delta))|$. Because for any bound dynamic variable δ_2 of σ_2, there must be a bound dynamic variable δ_1 of σ_1 such that $\delta_2 \in FV(S(\delta_1))$,

$$|\tau \leq \sigma_2| = \sum_{\delta \in BDV(\sigma_2)} |S_2(\delta)| \leq \sum_{\delta \in BDV(\sigma_1)} |S_2(S(\delta))| = |\tau \leq \sigma_1|$$

□

Size of full instance predicate $|\tau <: \mathcal{K}|$

The type size of instantiation $|\tau \leq \sigma|$ is rewritten to a form of $|\tau <: \mathcal{K}|$ by using *coerce*:

$$|\tau <: \mathcal{K}| = |\tau \leq coerce(\mathcal{K})|$$

To be well-defined, τ must be an instance of *coerce*$(\mathcal{K})$, but it need not to be a valid full instance type of $\mathcal{K}$.

8.6.2 Strictly admissible constraints for $\ll$:

Using the type size $|\cdot|$, we now define the set of the *strictly admissible constraints*. Roughly speaking, if a generic value is typed with a strictly admissible constraint, the sizes of run time type dispatching must *strictly* decrease in its recursive calls. This strict type size decreasing of recursive run time type dispatching ensures termination of recursion on types.

The following predicate *adm* defines the set of the strictly admissible constraints formally. When the constraint is a variable κ, instead of its μ-expansion, it returns *true* immediately when the corresponding recursive constraint $\mu\kappa.\mathcal{K}$ has been checked already. Otherwise *adm* returns *false*. For this purpose, the predicate takes an argument of a constraint substitution ψ, which keeps track of checkings of recursive constraints:

$$\begin{array}{rcl}
adm(\psi, [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]) & = & \forall i,j,\ \ adm(\psi, \psi(\mathcal{K}_{ij})) = true \\
 & & \quad \text{if } Dom(\psi) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset \\
 & & \quad \text{then } |\tau_i <: [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]| > |\tau_{ij} <: \psi(\mathcal{K}_{ij})| \\
adm(\psi, \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & adm(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \mathcal{K}) \\
adm(\psi, \kappa) & = & \kappa \in Dom(\psi)
\end{array}$$

A constraint $\mathcal{K}$ is *strictly admissible* if and only if $adm(\emptyset, \mathcal{K}) = true$. Since *adm* does not expand μ-abstractions, its termination is trivial.

Lemma 24 (Substitution preserves strict admissibility) *Any type substitution S preserves the admissibility of constraints: if $adm(\psi, \mathcal{K}) = true$, then $adm(\psi, S(\mathcal{K})) = true$.*

PROOF: (Sketch) The part S really affects is in the case of $adm(\psi, \mathcal{K})$ which compares the sizes of predicates $\tau <: \mathcal{K}$. From Lemma 22, this type ordering is kept by any substitution S. □

Lemma 25 (μ-expansion preserves strict admissibility) *Let $\mathcal{K}, \mathcal{K}'$ be constraints, ψ and φ be μ-expansion constraint substitutions and κ be a constraint variable, such that*

$$\begin{array}{l} \kappa \notin Dom(\psi) \\ \varphi = \{\kappa \Rightarrow \mathcal{K}\} \\ (\psi + \varphi) \circ (\psi + \varphi) = \psi + \varphi \end{array}$$

Then, if $adm(\psi + \varphi, \mathcal{K}) = true$ and $adm(\psi + \varphi, \mathcal{K}') = true$, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = true$

PROOF: By induction over the constraint $\mathcal{K}$.

Constraint variable If $\mathcal{K}' = \kappa$ then $adm(\psi+\varphi, \varphi(\kappa)) = adm(\psi+\varphi, \mathcal{K}) = true$. Otherwise, when $\mathcal{K} = \kappa' \neq \kappa$, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = adm(\psi + \varphi, \mathcal{K}') = true$.

Type patterns Let $\mathcal{K}' = [\forall V_i.\tau_i \ \{\tau_{ij} <: \mathcal{K}_{ij}\}]$. From $adm(\psi + \varphi, \mathcal{K}') = true$, for all i and j, we have

$$\begin{array}{l} adm(\psi + \varphi, \mathcal{K}_{ij}) = true \\ \text{if } (Dom(\psi) + \{\kappa\}) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset \text{ then } |\tau_i <: \mathcal{K}'| > |\tau_{ij} <: (\psi + \varphi)(\mathcal{K}_{ij})| \end{array}$$

From the induction hypothesis, $adm(\psi + \varphi, \varphi(\mathcal{K}_{ij})) = true$.

$KFV(\varphi(\mathcal{K}_{ij})) = KFV(\mathcal{K}_{ij}) \setminus \{\kappa\} \cup KFV(\mathcal{K})$. Since $\psi + \varphi$ is idempotent, $Dom(\psi + \varphi) \cap KFV(\mathcal{K}) = \emptyset$. Therefore,

$$\begin{array}{l} (Dom(\psi + \{\kappa\}) \cap KFV(\varphi(\mathcal{K}_{ij})) \\ = (Dom(\psi) + \{\kappa\}) \cap (KFV(\mathcal{K}_{ij}) \setminus \{\kappa\} \cup KFV(\mathcal{K})) \\ = (Dom(\psi) + \{\kappa\}) \cap (KFV(\mathcal{K}_{ij}) \setminus \{\kappa\}) \\ \subset (Dom(\psi) + \{\kappa\}) \cap KFV(\mathcal{K}_{ij}) \end{array}$$

This means, if $Dom(\psi + \varphi\} \cap KFV(\varphi(\mathcal{K}_{ij})) \neq \emptyset$, $Dom(\psi + \varphi) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset$ and therefore $|\tau_i <: \mathcal{K}'| > |\tau_{ij} <: (\psi + \varphi)(\mathcal{K}_{ij})|$. From the idempotency of $\psi + \varphi$, the right hand side of the inequation is equal to $|\tau_{ij} <: (\psi + \varphi)(\varphi(\mathcal{K}_{ij}))|$.

From these, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = true$.

Recursive constraints Let $\mathcal{K}' = \mu\kappa'.\mathcal{K}''$. Then $adm(\psi+\varphi, \mu\kappa'.\mathcal{K}'') = adm(\psi'', \mathcal{K}'') = true$, where

$$\begin{array}{l} \psi' = (\psi + \varphi) \setminus \{\kappa' \Rightarrow (\psi + \varphi)(\kappa')\} \\ \psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}'')\} \end{array}$$

$\psi'' \circ \psi'' = \psi''$ is easily verified.

Further proof depends on where κ' is κ or not:

When $\kappa' \neq \kappa$
$\psi' = (\psi + \varphi) \setminus \{\kappa' \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}'')\} + \varphi$.

$adm(\psi'', \mathcal{K}) = true$ is verified since $\mathcal{K} = \varphi(\kappa)$ and the idempotency of the constraint substitutions; the modification to the binding of κ' never affects the admissibility of $\mathcal{K}$, therefore $adm(\psi''.\mathcal{K}) = adm(\psi + \varphi, \mathcal{K}) = true$.

From induction hypothesis, we have $adm(\psi'', \varphi(\mathcal{K}'')) = true$.

From the definition and idempotency of ψ' and φ, $\psi'(\mathcal{K}'') = \psi'(\varphi(\mathcal{K}''))$. Therefore, $\psi'' = \psi\{\kappa' \Rightarrow \mu\kappa'.\psi'(\varphi(\mathcal{K}''))\} + \varphi$. From $adm(\psi'', \varphi(\mathcal{K}'')) = true$ and this, we obtain

$$adm(\psi + \varphi, \mu\kappa'.\varphi(\mathcal{K}'')) = adm(\psi + \varphi, \varphi(\mu\kappa'.\mathcal{K}'')) = true$$

When $\kappa' = \kappa$
$adm(\psi + \varphi, \varphi(\mu\kappa'.\mathcal{K}'')) = adm(\psi + \varphi, \mu\kappa'.\mathcal{K}'') = true.$

□

A generic constraint $\mathcal{K}$ is *strictly admissible* if and only if $adm(\mathcal{K}) = true$. Since *adm* does not expand μ-abstractions in its recursive calls, its termination is trivial.

Theorem 26 *$\forall\kappa \in Dom(\psi), adm(\psi, \psi(\kappa)) = true$, and $\psi \circ \psi = \psi$,*

Theorem 27 (Strict admissibility assures the termination of $kinst(\psi, \tau \ll: \mathcal{K})$) *Let τ be a type expression, $\mathcal{K}$ be a generic constraint, ψ be a constraint substitution such that $\psi \circ \psi = \psi$.*

If $adm(\psi, \mathcal{K}) = true$, $kinst(\psi, \tau \ll: \mathcal{K})$ always terminates.

Proof: By induction over the algorithm, we prove that type size $|\tau <: \psi(\mathcal{K})|$ strictly decreases at the case of type patterns of the recursive calls of $kinst(\psi, \tau \ll: \mathcal{K})$, when $\mathcal{K}$ has constraint variables which can be μ-expanded by ψ. In other case, we prove sizes of predicates are preserved.

Because the size of predicate $|\tau <: \mathcal{K}| > 0$ for any type,this assures the termination of the algorithm *kinst*.

Type patterns Let $\mathcal{K} = [\forall V_i.\tau_i \ \{\tau_{ij} <: \mathcal{K}_{ij}\}]$. If there is no i such that $\tau \leq \forall V_i.\tau_i$, *kinst* immediately returns *false*.

Otherwise, let p is the first index such that $\tau \leq \forall V_p.\tau_p$. We let S be the instantiation $\tau \leq_S \forall V_p.\tau_p$. Then $kinst(\psi, \tau \ll: \mathcal{K}) = \forall j, kinst(\psi, S(\tau_{pj}) \ll: S(\mathcal{K}_{pj}))$.

From $adm(\psi, \mathcal{K}) = true$, we have the following:

$$adm(\psi, \psi(\mathcal{K}_{pj})) = true$$
$$\text{if } Dom(\psi) \cap KFV(\mathcal{K}_{pj}) \neq \emptyset \text{ then } |\tau_p <: \mathcal{K}| > |\tau_{pj} <: \psi(\mathcal{K}_j)|$$

From Lemma 22, the inequation is rewritten as follows:

$$|\tau <: \mathcal{K}| > |S(\tau_{pj}) <: S(\psi(\mathcal{K}_j))|$$

Since $Dom(S) = V_p$, by renaming V_p, we can suppose $S(\psi(\mathcal{K}_j)) = \psi(S(\mathcal{K}_j))$. $\mathcal{K}$ is a type pattern constraint, ψ does not affect the type size of the predicate $|\tau <: \mathcal{K}|$: $|\tau <: \mathcal{K}| = |\tau <: \psi(\mathcal{K})|$. From these,

$$|\tau <: \psi(\mathcal{K})| > |S(\tau_{pj}) <: \psi(S(\mathcal{K}_j))|$$

Since $adm(\psi, \mathcal{K}_{pj}) = true$, from Lemma 24, $adm(\psi, S(\mathcal{K}_{pj}))$. We can use the induction hypothesis for $kinst(\psi, S(\tau_{pj}) \ll: S(\mathcal{K}_{pj}))$ for all j and assures their termination.

Recursive constraints Let $\mathcal{K} = \mu\kappa.[k_i]$. $kinst(\psi, \tau \ll: \mu\kappa.[k_i]) = kinst(\psi'', \tau \ll: [k_i])$, where $\psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi'\{\kappa \Rightarrow \mu\kappa.\psi'([k_i])\}$. Since

$$coerce(\psi(\mu\kappa.[k_i])) = coerce(\psi(\mu\kappa.\psi''[k_i])) = coerce([k_i])$$

The type size of the arguments of $kinq$ does not change: $|\tau <: \psi(\mu\kappa.[k_i])| = |\tau <: \psi''([k_i])|$.

From the idempotency of ψ, ψ' is also idempotent, therefore $\psi'\{\kappa \Rightarrow \mu\kappa.\psi(\mathcal{K}')\}$, too.

From $adm(\psi, \mu\kappa.[k_i]) = true$, $adm(\psi'', [k_i]) = true$. From Lemma 25, $adm(\psi'', \psi'[k_i]) = true$. This is equal to:

$$\begin{aligned} adm(\psi'', \psi'([k_i])) &= adm(\psi'', \mu\kappa.\psi'([k_i])) \\ &= adm(\psi'', \psi''(\kappa)) \end{aligned}$$

For constraint variables $\kappa' \neq \kappa$, $adm(\psi'', \psi''(\kappa')) = adm(\psi, \psi(\kappa')) = true$ is easily checked. Therefore $\forall \kappa' \in Dom(\psi''), adm(\psi'', \psi''(\kappa')) = true$ and we can use the induction hypothesis for the termination of $kinst(\psi'', \tau \ll: [k_i])$.

Constraint variables Let $\mathcal{K} = \kappa$. From $adm(\psi, \kappa) = true$, we have $\kappa \in Dom(\psi)$ and $\psi(\kappa)$ is well defined. Therefore, $kinst(\psi, \tau \ll: \kappa) = kinst(\psi, \tau \ll: \psi(\kappa))$. Since ψ is idempotent, $|\tau <: \psi(\kappa)| = |\tau <: \psi(\psi(\kappa))|$. Since $adm(\psi, \psi(\kappa)) = true$, we can use the induction hypothesis for $kinst(\psi, \tau \ll: \psi(\kappa))$ and obtain its termination.

$\square$

The following theorem is a special case of Theorem 27:

Theorem 28 (Strict admissibility assures the termination of $kinst(\emptyset, \tau \ll: \mathcal{K})$) *Let τ be a type expression and $\mathcal{K}$ be a generic constraint. If $adm(\mathcal{K}) = true$, then $kinst(\emptyset, \tau \ll: \mathcal{K})$ terminates.*

PROOF: This is a case of $\psi = \emptyset$ of Theorem 27 □

Thus, restricting constraints to admissible ones, such that $adm(\mathcal{K}) = true$, we make the predicate $\tau \ll: \mathcal{K}$ decidable and $kinst(\emptyset, \tau \ll: \mathcal{K})$ is the calculation for the predicate.

8.7 Relaxed full instances

The strict full instance relationship defined by construction using the constraint instantiation rules (k-case), (k-rec) and (k-derive) in the previous section is actually too strict in practice. We find difficulties when we define recursive derived generic values. Since their constraints cannot be strictly admissible, under the strict admissible constraint restriction, they cannot be defined in the type level. Therefore we must find a more relaxed restriction of constraints which permits derived recursive definitions, but still ensures the decidability of the full instance relationship.

8.7.1 Recursive let generics are not strictly admissible

Suppose we define a generic multiplication function `mult` for integers and floats:

```
generic mult = case
| int → int → int ⇒ ( * )
| float → float → float ⇒ ( *. )
;;
val mult : [int → int → int | float → float → float]
```

Then, define a recursive derived generic function `power` using this `mult` as follows:

```
let rec power n = function (* calculates pⁿ *)
| 1 → n
| p → mult n (power n (p - 1))
;;
```

Because `power` has a recursive occurrence of itself inside the definition with the type annotation $\delta \to \texttt{int} \to \delta$, `power` will have the following recursive constraint:

$$\mathcal{K}_{\texttt{power}} = \mu\kappa.[\forall\delta.\delta \to \texttt{int} \to \delta \ \{\delta \to \texttt{int} \to \delta <: \kappa\}]$$

But as far as we use $\lll:$ as full instance relationship, this function cannot be instantiated to any type. For example, $\texttt{int} \to \texttt{int} \to \texttt{int} \lll: \mathcal{K}_{\texttt{power}}$ is not provable. In addition, the algorithm *kinst* which checks valid full instantiation does not terminate with this type:

$$
\begin{aligned}
& \mathit{kinst}(\emptyset, \texttt{int} \to \texttt{int} \to \texttt{int} \lll: \mathcal{K}_{\texttt{power}}) \\
& = \mathit{kinst}(\{\kappa \Rightarrow \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \lll: [\forall \delta.\delta \to \texttt{int} \to \delta \ \{\delta \to \texttt{int} \to \delta <: \kappa\}]) \\
& = \mathit{kinst}(\{\kappa \Rightarrow \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \lll: \kappa) \\
& = \mathit{kinst}(\{\kappa \Rightarrow \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \lll: \mathcal{K}_{\texttt{power}}) \\
& = \mathit{kinst}(\{\kappa \Rightarrow \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \lll: [\forall \delta.\delta \to \texttt{int} \to \delta \ \{\delta \to \texttt{int} \to \delta <: \kappa\}]) \\
& = \ldots \ \text{(loops forever)}
\end{aligned}
$$

The algorithm loops forever due to μ-expansions, therefore we cannot find it is valid or invalid. This `power` must be rejected by the system using $\lll:$, and actually $\mathcal{K}_{\texttt{power}}$ is not a strictly admissible.

We also have such an example in the recursive generic bindings:

```
generic rec add =
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| δ * δ → δ ⇒ fun (x,y) → add x y
;;
```

Another variant of `add` generic function. In this time, the function is also usable in an uncurried form: `add (1,2)`, `add (1.0, 2.3)` and so on. The constraint of `add`, $\mathcal{K}_{\texttt{add}}$ is:

$$
\mathcal{K}_{\texttt{add}} = \mu\kappa. \quad
\begin{array}{l}
[\ \texttt{int} \to \texttt{int} \to \texttt{int} \\
\mid \texttt{float} \to \texttt{float} \to \texttt{float} \\
\mid \forall \delta.\delta * \delta \to \delta \ \{\delta \to \delta \to \delta <: \kappa\}]
\end{array}
$$

In the last type case, the internal `add` has a type annotation $\delta \to \delta \to \delta$. Therefore,

$$
|\delta{*}\delta \to \delta <: \mathcal{K}_{\texttt{add}}| = |\delta \to \delta \to \delta <: \mathcal{K}_{\texttt{add}}| = 3x + 1
$$

which means that $\mathit{adm}(\emptyset, \mathcal{K}_{\texttt{add}}) = \mathit{false}$, this constraint of `add` is not strictly admissible.

This situation is undesirable, since generally speaking, *all* the recursive derived generic values defined by let expressions and some by generic expressions do not have strictly admissible constraints, even when they have no risk of generic run time errors. This problem is due to over-restricted rules of the full instance relationship $\lll:$. The predicate $\tau \lll: \mathcal{K}$ assures not only the soundness of type recursion concerned with this predicate, but also the termination of the type recursion. What we actually want is only the first: the soundness. The termination is not necessary. Generic values which will never terminate should also be welcome if we can be sure they never fail, just like well-typed recursive functions may never terminates in normal ML. Note also that non-termination of type recursion does not mean non-termination of the evaluation directly. For example, even though type recursion of `power` function loops forever, if we give $\texttt{n} > 0$, it terminates and returns a result.

8.7.2 Instances as the complement of the non-instances

Instead of $\ll:$, we define another full instance relationship $<:$, as the complement of the *invalid full instances* which have possibility of failure. We call $<:$ the *relaxed full instance relationship.*

For the definition of $<:$, the *invalid full instance relationship* $\tau \not<: \mathcal{K}$ must be defined first:

$$\textsf{(k-fail-case)}\ \frac{\forall 1 \le i \le n,\ \tau \not\le \sigma_i \quad \text{or} \quad \forall 1 \le i \le p-1,\ \tau \not<: \sigma_i,\ \tau \le \sigma_p,\ \tau \not<: \sigma_p\ D_p}{\tau \not<: [\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]}$$

$$\textsf{(k-fail-rec)}\ \frac{\tau \not<: \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}}{\tau \not<: \mu\kappa.\mathcal{K}}$$

$$\textsf{(k-fail-derive)}\ \frac{\tau \not\le \sigma, \quad \text{or} \quad \exists 1 \le i \le n, \quad S(\tau_i) \not<: S(\mathcal{K}_i)\ \text{where}\ \tau \le_S \sigma}{\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n}$$

$$\textsf{(k-fail-free)}\ \tau \not<: \kappa$$

The rules are the negative versions of $\ll:$ rules and additional one for free constraint variables. The *relaxed full instance relationship*, $\tau <: \mathcal{K}$ is defined as the complement of $\tau \not<: \mathcal{K}$:

Definition 17 (Relaxed full instance) *Let τ be a type, $\mathcal{K}$ is a generic constraint. Then we say τ is a relaxed full instance (or just an instance) of $\mathcal{K}$ and write $\tau <: \mathcal{K}$, if and only if $\tau \not<: \sigma\ \mathcal{K}$ is not provable.*

The algorithm *kinst* is modified to provide an algorithm to determine $\tau <: \mathcal{K}$ or not, The new algorithm $kinst^+$ takes a new memo function argument M which remembers the μ-abstraction tests already it has tried. If it is asked the same question stored in the cache M, it immediately returns *true*:

$$\begin{array}{rcl}
kinst^+(M, \psi, \tau <: [\sigma_i\ D_i]_{i=1}^n) & = & \exists p, \text{where } \tau \not\le \sigma_1\ \&\ \ldots \tau \not\le \sigma_{p-1}\ \&\ \tau \le \sigma_p \\
 & & \&\ kinst^+(M, \psi, \tau <: \sigma_p\ D_p) \\
kinst^+(M, \psi, \tau <: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n) & = & \tau \le_S \sigma\ \&\ \forall_{i=1}^n kinst^+(M, \psi, S(\tau_i) <: S(\mathcal{K}_i)) \\
kinst^+(M, \psi, \tau <: \mu\kappa.\mathcal{K}) & = & \text{if } (\tau <: \mu\kappa.\mathcal{K}) \in M \text{ then } true \\
 & & \text{else let } M' = M \cup \{\tau <: \mu\kappa.\mathcal{K}\} \\
 & & \quad \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & \quad kinst^+(M', \psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \tau <: \mathcal{K}) \\
kinst^+(M, \psi, \tau <: \kappa) & = & \text{if } \kappa \notin Dom(\psi) \text{ then } false \\
 & & \text{else } kinst^+(M, \psi, \tau <: \psi(\kappa))
\end{array}$$

Theorem 29 (Correctness of $kinst^+$ against $<:$) *Let τ be a type and $\mathcal{K}$ be a constraint. Suppose $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ terminates. Then $\tau \not<: \mathcal{K}$ if and only if $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ returns false.*

PROOF: (Sketch) The proof follows the same outline of Theorem 25. Suppose $\tau \not<: \mathcal{K}$. then we can check easily $kinst^+(\emptyset, \emptyset, \tau \not<: \mathcal{K}) = false$ by induction over the derivation of $\tau \not<: \mathcal{K}$. Only thing we should note that while the evaluation of the algorithm, the cached predicate in the memo function M is never used. Otherwise we would have the exactly same judgments in a derivation path:

$$\cfrac{\cfrac{\cfrac{\cdots}{\tau \not<: \sigma\ \mu\kappa.\mathcal{K}}}{\cdots}}{\cfrac{\tau \not<: \sigma\ \mu\kappa.\mathcal{K}}{\cdots}}$$

which cannot happen, because the derivation would become infinite.

Contrary, suppose $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K}) = false$. We can build the proof of $\tau \not<: \mathcal{K}$ by induction over the evaluation of the algorithm $kinst^+$. □

Same as the strict relationship, we need to restrict the constraints to something admissible in order to make the predicate $\not<:$ decidable and so does $<:$, in order to $kinst^+$ can terminate for any restricted constraint.

8.7.3 Relaxed admissible constraints for $<:$

Like the strictly admissible constraints which were defined for the termination of $kinst(\tau \ll: \mathcal{K})$, we now define the set of *the relaxed admissible constraints* for $kinst^+(\tau <: \mathcal{K})$. The predicate adm^+ which defines the set of relaxed admissible constraints differs from adm in only one point: the type sizes also must be decreasing in the recursive part of the derived constraints in adm^+, but the strict decrease is not required:

$$\begin{array}{rcl}
adm^+(\psi, [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]) & = & \forall i, j,\ \ adm^+(\psi, \psi(\mathcal{K}_{ij})) = true \\
 & & \qquad \text{if } Dom(\psi) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset \\
 & & \qquad \text{then } |\tau_i <: [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]| \geq |\tau_{ij} <: \psi(\mathcal{K}_{ij})| \\
adm^+(\psi, \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & adm^+(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \mathcal{K}) \\
adm^+(\psi, \kappa) & = & \kappa \in Dom(\psi)
\end{array}$$

A constraint $\mathcal{K}$ is *relaxed admissible* if and only if $adm^+(\emptyset, \mathcal{K})$ holds. Clearly, from this definition, all the strictly admissible constraints are also relaxed admissible.

To prove this restriction assures the termination of $kinst^+$, first we need some lemmas, just like we have done in Section 8.6.2:

Lemma 26 (Substitution preserves relaxed admissibility) *Any type substitution S preserves the relaxed admissibility of constraints: if $adm^+(\psi, \mathcal{K}) = true$, then $adm^+(\psi, S(\mathcal{K})) = true$.*

PROOF: Similar to Lemma 24. □

Lemma 27 (μ-expansion preserves relaxed admissibility) *Let $\mathcal{K}, \mathcal{K}'$ be constraints, ψ and φ be μ-expansion constraint substitutions and κ be a constraint variable, such that*

$$\begin{array}{l} \kappa \notin Dom(\psi) \\ \varphi = \{\kappa \Rightarrow \mathcal{K}\} \\ (\psi + \varphi) \circ (\psi + \varphi) = \psi + \varphi \end{array}$$

Then, if $adm^+(\psi + \varphi, \mathcal{K}) = true$ and $adm^+(\psi + \varphi, \mathcal{K}') = true$, $adm^+(\psi + \varphi, \varphi(\mathcal{K}')) = true$

PROOF: Similar to Lemma 25. □

Lemma 28 (Dynamic substitution preserves relaxed admissibility) *Type substitutions whose domains are restricted within the dynamic type variables preserve the admissibility of type schemes: for a substitution S such that $Dom(S) \subset \mathbb{D}$, if $adm^+(\emptyset, \eta) = true$, then $adm^+(\emptyset, S(\eta)) = true$.*

PROOF: Same as Lemma 24. □

Theorem 30 (Relaxed admissibility assures the termination of $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$) *Let $\mathcal{K}$ be a relaxed admissible extended constraint. Then for any type expression τ, $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ always terminates.*

PROOF: The termination can be proven almost similarly to one for *kinst*, Theorem 27. The difference is the cases when the sizes of $|\tau <: \mathcal{K}|$ do not change and never decrease. Even in such cases, since we have only finite type constructors, the predicates $|\tau <: \mathcal{K}|$ which share the same size are finite. Therefore sooner or later $kinst^+(M, \psi, \tau <: \mathcal{K})$ will find $\tau <: \mathcal{K}$ is stored in the cache M and terminate. □

Using the relaxed admissible constraint restriction, all the let defined recursive derived generic values and many useful recursive values defined by generic expressions such as `power` and `add` in previous examples are now admissible and permitted in type system.

8.8 Polymorphic instances

The full instances are not only the permitted generic instances in constraint type system, otherwise we could not define any derived and recursive generic value. *Polymorphic instances* are also allowed, intuitively whose type annotations are abstracted somewhere outside. Validness of polymorphic instances are postponed until the derived generic values which use these polymorphic instances are instantiated to full instances. Therefore type information of polymorphic instances are stored in the derived generic values' constraints as form of derived constraints. Before we define the constraint typing rules, we have to give the formal definition of the full and polymorphic instances.

When dynamic type variables of type annotation of a generic instance is generalized, the instance is a polymorphic instance. Such polymorphic instances are found in definitions of derived generic values. For example, the following instance of `print` generic value:

```
let print_twin x =
  print (x,x)
in ...
```

The `print` instance has a type $\delta * \delta \to \texttt{unit}$ where the type variable δ is abstracted at the generalization of `print_twin`, whose constraint is $[\forall\delta.\delta \to \texttt{unit}\ \{\delta * \delta \to \texttt{unit} <: \mathcal{K}_{\texttt{print}}\}]$. The type information of the polymorphic instance `print` is stored as the derived constraint $\{\delta * \delta \to \texttt{unit} <: \mathcal{K}_{\texttt{print}}\}$.

A generic instance may be a polymorphic instance even when its type does not contain any generalized dynamic type variable. If the constraint of a generic value contains free dynamic type variable which is generalized at some generalization, the instances of the generic value is polymorphic instances. Suppose `plus` is a generic value whose constraint is

$$\begin{aligned}\mathcal{K}_{\texttt{plus}} = \ & [\ \texttt{int} \to \texttt{int} \to \texttt{int} \\ & \mid \texttt{int} \to \texttt{float} \to \texttt{float} \\ & \mid \texttt{float} \to \texttt{int} \to \texttt{float} \\ & \mid \texttt{float} \to \texttt{float} \to \texttt{float}\ ]\end{aligned}$$

Then in the following example,

```
let f = fun x →
  let g = fun y →
    plus x y
  in
  print_int (g 1)
in ...
```

Let the type of `x` in the definition of `f` is δ. Then the constraint of `g` is

$$\mathcal{K}_{\texttt{g}} = [\forall\delta'\delta''.\delta' \to \delta''\ \{\delta \to \delta' \to \delta'' <: \mathcal{K}_{\texttt{plus}}\}]$$

which contains δ freely in its derived part and this δ is generalized at the definition of `f`. The type soundness of full instances of `f` depends on one for the instance of `g`, whose type is $\texttt{int} \to \texttt{int}$. The soundness of `g` depends on how δ is instantiated for each full instance of `f`. Actually, because of $\mathcal{K}_{\texttt{g}}$'s constraint, the only possible instantiation is $(\delta \Rightarrow \texttt{int})$ so that `plus` can have a type $\texttt{int} \to \texttt{int} \to \texttt{int}$. To express this dependency of type soundness, the generic instance of `g` must be a polymorphic instance, even though its type has no dynamic type variable. The type information of `g` must be exported as a derived constraint of `f`:

$$\mathcal{K}_{\texttt{f}} = [\forall\delta.\delta \to \texttt{unit}\ \{\texttt{int} \to \texttt{int} <: \mathcal{K}_{\texttt{g}}\}]$$

The predicate *polyinst* distinguishes polymorphic instances from the others — full instances. *polyinst* takes a set of type variables V which are generalized, a constraint $\mathcal{K}$ and a type τ:

$$\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{generic}(\mathcal{K})\ \&\ (DV(\tau) \cup DV(\mathcal{K})) \cap V \neq \emptyset$$

Then the polymorphic instances are defined as follows:

Definition 18 (Polymorphic instances) *Let* g *be an instance,* V *is the generalized type variable set at the occurrence of* g*, e.g. the type variables generalized by* let *or* generic *polymorphisms whose definition expression contains the instance* g*. If* g *has a constraint* $\mathcal{K}$ *and annotated* g : τ*, and* $polyinst(V, \mathcal{K}, \tau) = true$*,* g *is called a* polymorphic instance*.*

8.9 Generic constraint type system

Using the (relaxed) full instance relationship, we build the *constraint type system*. As we have explained in the introduction of this chapter, constraint type system which checks the validity of the generic instances is separated from the *normal typing system* which has been introduced in Section 2.5 to avoid confusion between these two typing layers. Constraint typing system is designed to receive type-annotated terms which have already been checked by the normal type system, and perform further checks whether all the generic full instances in the terms are valid full instances.

8.9.1 Constraint annotated terms

We first add constraints to the type-annotated terms (whose original definition is in Section 2.3 and extended for the generic bindings at Section 7.3). The new expressions are called *constraint annotated terms*.

$$
\begin{array}{rcl}
e_t & ::= & \mathtt{x}_{\mathcal{K}} : \tau \\
& | & c_\sigma : \tau \mid \mathtt{g}_\sigma : \tau \mid \mathtt{fun}\ \mathtt{x} \rightarrow e_t : \tau \mid e_t\ e_t : \tau \mid e_t, e_t : \tau \\
& | & \mathtt{let}\ \mathtt{x}_{V,\mathcal{K}} = e_t\ \mathtt{in}\ e_t : \tau \\
& | & \mathtt{let\ rec}\ \mathtt{x}_{V,\mathcal{K}} = e_t\ \mathtt{in}\ e_t : \tau \\
& | & \mathtt{generic}\ \mathtt{x}_{\mathcal{K}} = \mathtt{case}\ tc_t \mid \ldots \mid tc_t\ \mathtt{in}\ e_t : \tau \\
& | & \mathtt{generic\ rec}\ \mathtt{x}_{\mathcal{K}} = \mathtt{case}\ tc_t \mid \ldots \mid tc_t\ \mathtt{in}\ e_t : \tau \\
tc_t & \equiv & \sigma \Rightarrow e_t
\end{array}
$$

Variables now take constraints $\mathcal{K}$ instead of type schemes. Let bindings and generic bindings take their constraints $\mathcal{K}$ additional to the original annotations of identifiers.

The type annotation substitution is canonically extended:

$$
\begin{array}{l}
S(\mathtt{x}_{\mathcal{K}} : \tau) = \mathtt{x}_{S(\mathcal{K})} : S(\tau) \\
S(\mathtt{let\ (rec)}\ \mathtt{x}_{V,\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}) = \mathtt{let\ (rec)}\ \mathtt{x}_{V,S(\mathcal{K})} = S(e_{t1})\ \mathtt{in}\ S(e_{t2}) \\
S(\mathtt{generic\ (rec)}\ \mathtt{x}_{\mathcal{K}} = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t) = \\
\qquad\qquad \mathtt{generic\ (rec)}\ \mathtt{x}_{S(\mathcal{K})} = \mathtt{case}\ S(tc_{t1}) \mid \ldots \mid S(tc_{tn})\ \mathtt{in}\ S(e_t) \\
S(\sigma \Rightarrow e_t) = (\sigma \Rightarrow S(e_t))
\end{array}
$$

The definitions for the other constructions are as they were, because all their type annotations are same as the original syntax therefore have no constraints. Note that generalized variable set V in a let binding bounds type variable occurrences in the constraint annotation $\mathcal{K}$ also. If renaming of the bound variables is required for substitution, we also have to rename variables in $\mathcal{K}$.

Extending the coercion from constraints to types *coerce* defined at Section 8.4.5, we obtain the following mapping from constraint annotated terms to the original type-annotated GML expressions:

$$
\begin{array}{l}
\mathit{coerce}(\mathtt{x}_{\mathcal{K}} : \tau) = \mathtt{x}_{\mathit{coerce}(\mathcal{K})} : \tau \\
\mathit{coerce}(c_\sigma : \tau) = c_\sigma : \tau \\
\mathit{coerce}(\mathtt{g}_\sigma : \tau) = \mathtt{g}_\sigma : \tau \\
\mathit{coerce}(\mathtt{fun\ x} \rightarrow e_t : \tau) = (\mathtt{fun\ x} \rightarrow \mathit{coerce}(e_t)) : \tau \\
\mathit{coerce}(e_{t1}\ e_{t2} : \tau) = \mathit{coerce}(e_{t1})\ \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(e_{t1}, e_{t2} : \tau) = \mathit{coerce}(e_{t1}), \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(\mathtt{let\ (rec)\ x}_{V,\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \mathtt{let\ (rec)\ x}_V = \mathit{coerce}(e_{t1})\ \mathtt{in}\ \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(\mathtt{generic\ (rec)\ x}_{\mathcal{K}} = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t : \tau) = \\
\qquad \mathtt{generic\ (rec)\ x}_{\mathit{coerce}(\mathcal{K})} = \mathtt{case}\ \mathit{coerce}(tc_{t1}) \mid \ldots \mid \mathit{coerce}(tc_{tn})\ \mathtt{in}\ \mathit{coerce}(e_t) : \tau \\
\mathit{coerce}(\sigma \Rightarrow e_t) = (\sigma \Rightarrow \mathit{coerce}(e_t))
\end{array}
$$

8.9.2 Constraint typing rules

A typing judgment in the constraint typing system takes a form:

$$\Gamma, V \Vdash e_t$$

where e_t is a constraint annotated term defined just before and Γ is a typing environment where variables are bound to constraints. Because type instantiations of generic values are differently handled if instances are polymorphic instances or not, the judgment also carries V, a *generalized type variable set*, which indicates which type variables must be generalized in following typing derivation. In effect, the free occurrences of dynamic type variables in e_t are restricted within those of V.

Instance rules

We have two instantiation rules: one for the full and another for the polymorphic instances. (kt-full-inst) is for full instantiations:

$$\textbf{(kt-full-inst)}\ \frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{false} \quad \tau <: \mathcal{K}}{\Gamma, V \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

To use this rule, the instance of `x` must be fully instantiated. The dynamic type variables in τ are restricted to those of V.

If a variable occurrence is a polymorphic instance, (kt-full-inst) is not applicable, instead, (kt-poly-inst) must be used even if $\tau <: \mathcal{K}$ holds:

$$\textbf{(kt-poly-inst)}\ \frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}}{\Gamma, V \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

If $\mathtt{x}$ is a polymorphic instance, *any* type τ can be the type of $\mathtt{x}$, only if its dynamic type variables are members of the generalized variables V. We need neither check whether τ is an instance of the type scheme of $\mathtt{x}$, $\mathit{p\_antiunif}(\mathcal{K})$, because we suppose that this has been proven already in the normal typing layer.

Constraint construction and rules for polymorphic bindings

Corresponding with generalizations of type expressions in the normal typing system, we create constraints in the let and generic bindings in the constraint typing. The constraint construction function $\mathit{Derive}(V, V', e_t)$ in Figure 8.1 takes two sets of type variables V and V' and a type-annotated term e_t, and returns a derived constraint. It based on function $\mathit{plist}(V, V', e_t)$ which lists up the polymorphic instance occurrences $\mathtt{x}_{\mathcal{K}} : \tau$ in e_t whose type annotations relate with V but not with V'. (The symbol @ is used for the list concatenation in the figure). This exclusion is not essentially required for the type soundness itself, but necessary not to store one polymorphic instance information into more than one constraints. Type annotation information of a polymorphic instance is stored into a derived constraint created at the most interior generalization which abstract dynamic type variables of the polymorphic instance, so that we can minimize the complexity of constraints (look at the next example). *Derive* function extracts type annotations from result of $\mathit{plist}\ [\ \mathtt{x}^1_{\mathcal{K}_1} : \tau_1;\ \ldots;\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\ ]$ and forms a derived constraint $\{\tau_1 <: \mathcal{K}_1 \ \& \ldots \&\ \tau_n <: \mathcal{K}_n\}$. The order of the derived constraints $\tau_i <: \mathcal{K}_i$ must corresponds with the occurrence order of those polymorphic instances[1].

Let bindings

Using *Derive*, constraint typing rules for polymorphic bindings can be defined:

$$\textbf{(kt-let)}\ \frac{\begin{array}{c}\Gamma, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

The constraint typing rule for let bindings checks soundness of e_{t1} under $\Gamma, V + V'$, adding the generalized variable set V' of the conclusion annotation. For e_{t2}, the environment Γ is updated with a binding $\mathtt{x} : \mathcal{K}$ where $\mathcal{K} = [\forall V'.\tau_1\ D]$. The derived constraint D is the result of the constraint construction $\mathit{Derive}(V', \emptyset, e_{t1})$. To assure the termination of typing, the constraint $\mathcal{K}$ must be relaxed admissible. This admissible restriction is required also in the following rules which construct constraints.

[1] Minimization and fixed order of derived constraints are required for the correctness of generic flow compilation. Look Chapter 9 for details.

$$
\begin{array}{l}
\mathit{plist}(V, V', c_\sigma : \tau) = [\,] \\
\mathit{plist}(V, V', \mathtt{g}_\sigma : \tau) = [\,] \\
\mathit{plist}(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = \text{ if } \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true} \;\&\; \mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false} \\
\qquad\qquad\qquad\qquad\quad \text{then } [\ \mathtt{x}_{\mathcal{K}} : \tau\ ] \\
\qquad\qquad\qquad\qquad\quad \text{else } [\,] \\
\mathit{plist}(V, V', e_{t1}\ e_{t2}) = \mathit{plist}(V, V', e_{t1}) \mathbin{@} \mathit{plist}(V, V', e_{t2}) \\
\mathit{plist}(V, V', (e_{t1},\ e_{t2})) = \mathit{plist}(V, V', e_{t1}) \mathbin{@} \mathit{plist}(V, V', e_{t2}) \\
\mathit{plist}(V, V', \mathtt{fun}\ \mathtt{x} \rightarrow e_t) = \mathit{plist}(V, V', e_t) \\
\mathit{plist}(V, V', \mathtt{let}\ (\mathtt{rec})\ \mathtt{x}_{V'', \mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}) = \mathit{plist}(V, V' \cup V'', e_{t1}) \mathbin{@} \mathit{plist}(V, V', e_t) \\
\mathit{plist}(V, V', \mathtt{generic}\ (\mathtt{rec})\ \mathtt{x} = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_{t2}) = \\
\qquad\qquad\qquad\qquad\qquad \mathit{plist}(V, V', tc_{t1}) \mathbin{@} \ldots \mathbin{@} \mathit{plist}(V, V', tc_{tn}) \mathbin{@} \mathit{plist}(V, V', e_t) \\
\mathit{plist}(V, V', (\forall V''.\tau \Rightarrow e_t)) = \mathit{plist}(V, V' \cup V'', e_t) \\
\\
\mathit{Derive}(V, V', e_t) = \text{ let } [\ \mathtt{x}^1_{\mathcal{K}_1} : \tau_1;\ \ldots;\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\ ] = \mathit{plist}(V, V', e_t) \text{ in} \\
\qquad\qquad\qquad\qquad \{\tau_1 <: \mathcal{K}_1 \;\&\ldots\&\; \tau_n <: \mathcal{K}_n\}
\end{array}
$$

Figure 8.1: Constraint computation function *Derive*

Example In the following type-annotated expression

$$
\begin{array}{l}
\mathtt{val\ f\ :}\ [\forall\delta.\delta \rightarrow \mathtt{unit}\ D_{\mathtt{f}}] \\
\mathtt{let}\ \mathtt{g}_{\{\delta'\},\mathcal{K}_{\mathtt{g}}}\ \mathtt{=\ fun\ x} \rightarrow \\
\quad \mathtt{let}\ \mathtt{h}_{\{\delta''\},\mathcal{K}_{\mathtt{h}}}\ \mathtt{=\ fun\ y} \rightarrow \\
\quad\quad \mathtt{(f:}\delta'\mathtt{*}\delta'' \rightarrow \mathtt{unit)\ (x,y)} \\
\quad \mathtt{in} \\
\quad \mathtt{(h:}\delta' \rightarrow \mathtt{unit)\ x} \\
\mathtt{in} \\
\quad \ldots
\end{array}
$$

suppose that the normal typing gives `g` a type scheme $\forall\delta'.\delta' \rightarrow \mathtt{unit}$ and `h` $\forall\delta''.\delta'' \rightarrow \mathtt{unit}$. In constraint typing, let the constraint of the generic function `f` be $\mathcal{K}_{\mathtt{f}} = [\forall\delta.\delta \rightarrow \mathtt{unit}\ D_{\mathtt{f}}]$.

`h`'s constraint $\mathcal{K}_{\mathtt{h}}$ must be as follows, because it contains a polymorphic occurrence of `f` inside the definition whose type annotation has type variable δ'' which is generalized by the definition of `h`:

$$
\begin{array}{l}
D_{\mathtt{h}} = \mathit{Derive}(\{\delta''\}, \emptyset, \mathtt{fun\ y} \rightarrow \mathtt{(f\ :}\ \delta' * \delta'' \rightarrow \mathtt{unit)\ (x,y)}) = \{\delta' * \delta'' \rightarrow \mathtt{unit} <: \mathcal{K}_{\mathtt{f}}\} \\
\mathcal{K}_{\mathtt{h}} = [\forall\delta''.\delta'' \rightarrow \mathtt{unit}\ D_{\mathtt{h}}]
\end{array}
$$

On the other hand, though the type variable δ' of the polymorphic instance of `f` is generalized by the definition of `g`, `f` is not directly taken into $\mathcal{K}_{\mathtt{g}}$, the constraint of `g`, because it is already listed in $\mathcal{K}_{\mathtt{h}}$ whose definition is interior to g for the instance f:

$$\begin{array}{l} D_{\mathtt{g}} = \mathit{Derive}(\{\delta'\}, \emptyset, \mathtt{fun\ x} \rightarrow \mathtt{let\ h = \dots\ in\ (h}_{\mathcal{K}_{\mathtt{h}}} : \delta' \rightarrow \mathtt{unit)\ x}) = \\ \qquad \mathit{Derive}(\{\delta'\}, \{\delta''\}, \mathtt{fun\ y} \rightarrow (\mathtt{f}_{\mathcal{K}_{\mathtt{f}}} : \delta' * \delta'' \rightarrow \mathtt{unit})\ (\mathtt{x}, \mathtt{y})) \mathbin{@} \\ \qquad\qquad \mathit{Derive}(\{\delta'\}, \emptyset, (\mathtt{h}_{\mathcal{K}_{\mathtt{h}}} : \delta' \rightarrow \mathtt{unit})\ \mathtt{x}) = \\ \qquad \{\delta' \rightarrow \mathtt{unit} <: \mathcal{K}_{\mathtt{h}}\} \\ \mathcal{K}_{\mathtt{g}} = [\forall \delta'.\delta' \rightarrow \mathtt{unit}\ D_{\mathtt{g}}] \end{array}$$

The constraint of g, $\mathcal{K}_{\mathtt{g}}$ does not contain the derived constraint of f directly, but its dependency on f is expressed $\mathcal{K}_{\mathtt{h}}$ in $D_{\mathtt{g}}$, since $\mathcal{K}_{\mathtt{h}}$ contains a derived constraint from f.

□

Recursive lets

The recursive counterpart of (kt-let), (kt-letrec) stores its identifier x to environment Γ with a recursive constraint $\mu\kappa.\mathcal{K}$ in typings of both sub-expressions:

$$\textbf{(kt-letrec)}\ \frac{\begin{array}{c} \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t2} \\ D' = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D'] \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\rightarrow} \mathcal{K}' \\ \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true} \end{array}}{\Gamma, V \Vdash \mathtt{let\ rec\ x}_{V', \mu\kappa.\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}}$$

The constraint $\mathcal{K}'$ which contains derived constraint D' obtained by $\mathit{Derive}(V', \emptyset, e_{t1})$ must be equal to the μ-expansion of $\mu\kappa.\mathcal{K}$: it must be a fixed point. In the typing of the let definition e_{t1}, x has a polymorphic constraint $\mu\kappa.\mathcal{K}$, unlike normal ML typing where a recursive let identifier must have a monomorphic type inside its definition. It corresponds with the fact that the variable occurrences of x in e_{t1} have type scheme annotations with $\forall V'.\tau_1$ in (t-letrec) typing rule (Section 2.5.3).

When e_{t1} contains no polymorphic instance occurrence, the derived constraint D' from e_{t1} is an empty set and $\mathcal{K}'$ is simply $[\forall V'.\tau_1]$. In such a non-generic recursive definition, the constraint of the defined value is actually non-recursive, since $\mu\kappa.\mathcal{K} = \mu\kappa.[\forall V'.\tau_1] = \mu\kappa.\mathcal{K}'$ and $\mu\kappa.\mathcal{K}' \equiv \mathcal{K}'$ from Definition 16.

Generic bindings

$$\textbf{(kt-generic)}\ \frac{\begin{array}{c} \Gamma, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \dots \mid \sigma_n\ D_n] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t \end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_{\mathcal{K}} = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \dots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

For generic bindings, the type system checks type method expressions e_{ti} and creates their derived constraints D_i like in (kt-let). In the check of the expression's body e_t, x is stored as $\mathtt{x} : [\sigma_1\ D_1 \mid \dots \mid \sigma_n\ D_n]$, with a list of combinations of the type pattern schemes and corresponding derived constraints.

Recursive generic bindings

Like (kt-letrec), a recursively defined identifier by `generic` expression has a recursive constraint $\mu\kappa.\mathcal{K}$ and this must be μ-expanded to $\mathcal{K}'$, the collection of the derived constraints $\sigma_i\ D_i$ of the typings of the case expressions.

$$\textsf{(kt-genericrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K}' = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\to} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\mu\kappa.\mathcal{K}}\,\mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

The others

The other rules, for constants, generic primitives, functions, applications and tuples are quite straightforward, except the restriction that they can only introduce dynamic type variables which are stored in their generalized type variable sets.

As a summary, we put all the constraint typing rules in Figure 8.2.

Example Let's think typing of a generic binding of `print` given by the following definition:

```
generic rec print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ → unit ⇒
    function [] → ()
           | x :: xs → print x; print xs
;;
```

Let the constraint of `print` be $\mathcal{K}$, which is still unsure for us. The typing of the first two type cases are trivial. Since the type pattern of the third case has an abstract type variable δ, it method is checked with the generalized variable set $\{\delta\}$:

$$\frac{\dfrac{\Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \mathtt{print} : \delta \to \mathtt{unit} \quad \Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \mathtt{print} : \delta\ \mathtt{list} \to \mathtt{unit}}{\ldots}}{\begin{array}{l}\Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \quad \mathtt{(function\ []} \to \mathtt{()} \\ \qquad\qquad\qquad\qquad\qquad \mathtt{\mid x{::}xs} \to \mathtt{print\ x;\ print\ xs)}\ : \delta\ \mathtt{list} \to \mathtt{unit}\end{array}}$$

Since the two internal `print` generic instances are typed with this generalized variable δ, they are polymorphic instances. In use of (kt-genericrec), their type informations are exported to the derived constraint of `print` and their validity checks are postponed to full instantiations of `print`:

$$\begin{array}{l}\mathit{Derive}(\{\delta\}, \emptyset,\ \mathtt{function\ []} \to \mathtt{\ldots;}\ \ \mathtt{print}_{\mathcal{K}}\ \mathtt{xs}\ ) = \\ \qquad\qquad\qquad\qquad \{\delta\ \to \mathtt{unit} <: \mathcal{K}\ \&\ \delta\ \mathtt{list} \to \mathtt{unit} <: \mathcal{K}\}\end{array}$$

$$\textbf{(kt-const)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash c_\sigma : \tau} \qquad \textbf{(kt-gen)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash \mathtt{g}_\sigma : \tau}$$

$$\textbf{(kt-fun)}\ \frac{DV(\tau_1) \subset V \quad \Gamma\{\mathtt{x} : [\tau_1]\}, V \Vdash e_t}{\Gamma, V \Vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2} \qquad \textbf{(kt-app)}\ \frac{\Gamma, V \Vdash e_{t1} \quad \Gamma, V \Vdash e_{t2}}{\Gamma, V \Vdash e_{t1}\ e_{t2}}$$

$$\textbf{(kt-tpl)}\ \frac{\Gamma, V \Vdash e_{t1} \quad \Gamma, V \Vdash e_{t2}}{\Gamma, V \Vdash (e_{t1},\ e_{t2})}$$

$$\textbf{(kt-full-inst)}\ \frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{false} \quad \tau <: \mathcal{K}}{\Gamma, V \Vdash \mathtt{x}_\mathcal{K} : \tau}$$

$$\textbf{(kt-poly-inst)}\ \frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}}{\Gamma, V \Vdash \mathtt{x}_\mathcal{K} : \tau}$$

$$\textbf{(kt-let)}\ \frac{\begin{array}{c}\Gamma, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

$$\textbf{(kt-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t2} \\ D' = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D'] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}' \\ \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

$$\textbf{(kt-generic)}\ \frac{\begin{array}{c}\Gamma, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_\mathcal{K} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

$$\textbf{(kt-genericrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K}' = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\mu\kappa.\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

Figure 8.2: Constraint typing rules

Gathering these three derived constraints of the type cases (actually the first two are empty), we construct a type case constraint $\mathcal{K}'$:

$$\begin{aligned}\mathcal{K}' = \; & [\texttt{int} \rightarrow \texttt{unit} \mid \texttt{string} \rightarrow \texttt{unit} \\ & \mid \forall \delta . \delta \; \texttt{list} \rightarrow \texttt{unit} \; \{ \; \delta \rightarrow \texttt{unit} <: \mathcal{K} \;\&\; \delta \; \texttt{list} \rightarrow \texttt{unit} <: \mathcal{K} \}]\end{aligned}$$

The constraint of `print` $\mathcal{K}$ must be μ-expanded to this $\mathcal{K}'$. Therefore:

$$\begin{aligned}\mathcal{K} = \mu\kappa . \; & [\texttt{int} \rightarrow \texttt{unit} \mid \texttt{string} \rightarrow \texttt{unit} \\ & \mid \forall \delta . \delta \; \texttt{list} \rightarrow \texttt{unit} \{ \; \delta \rightarrow \texttt{unit} <: \kappa \\ & \qquad\qquad\qquad\qquad \& \; \delta \; \texttt{list} \rightarrow \texttt{unit} <: \kappa \}]\end{aligned}$$

□

By the constraint typing rules, free dynamic variable occurrences in expressions are restricted:

Lemma 29 **(Dynamic variable closedness)** *Let Γ be a typing environment, V be a set of type variables and e_t be an expression which has a typing $\Gamma, V \Vdash e_t$. Then $DV(e_t) \subset V$ and $DV(\mathit{Derive}(V, \emptyset, e_t)) \subset V$*

Proof: Easily proven by induction over the constraint typing derivation. When rules introduce new dynamic type variables and create a judgment $\Gamma, V \Vdash e_t$, the dynamic variables are always restricted to those of the generalized type variable set V.

$DV(\mathit{Derive}(V, \emptyset, e_t)) \subset V$ is clear from the definition of *Derive*. □

Lemma 30 *The typing preserves well formedness of typing environments: if a conclusion judgment has a well formed typing environment Γ, then its all premise judgments also have well formed typing environments.*

Proof: Clear from the definition of the well formedness and typing rules.

□

Like the non-generic type soundness of the normal typing, a key to the *generic type soundness* which assures no generic run time type error is a substitution lemma, too.

Lemma 31 **(Substitution lemma of *Derive*)** *Let e_t be a constraint annotated expression and $D = \mathit{Derive}(V, V', e_t)$. Let S be a substitution such that V and V' are out of the reach of S. Then $\mathit{Derive}(V, V', S(e_t)) = S(D)$.*

Proof: By induction over the structure of e_t.

Variables Let e_{t1} be a variable $\mathtt{x}_{\mathcal{K}} : \tau$. Then $D = \mathit{Derive}(V, V', \mathtt{x}_{\mathcal{K}} : \tau)$.

If $\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}$ & $\mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false}$ then $D = \{\tau <: \mathcal{K}\}$. Because V and V' are out of S, S does not affect the free type variable occurrences of V and V' in τ and $\mathcal{K}$. Therefore, $\mathit{polyinst}(V, S(\mathcal{K}), S(\tau)) = \mathit{true}$ & $\mathit{polyinst}(V', S(\mathcal{K}), S(\tau)) = \mathit{false}$ and $\mathit{Derive}(V, V', S(\mathtt{x}_{\mathcal{K}} : \tau)) = \{S(\tau) <: S(\mathcal{K})\} = S(D)$.

Otherwise, if $\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}$ & $\mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false}$ is not true, $\mathit{Derive}(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = \{\}$. By the same reason above, we obtain $\mathit{Derive}(V, V', S(\mathtt{x}_{\mathcal{K}} : \tau)) = \{\}$.

Let bindings $e_t = \mathtt{let}\ \mathtt{y}_{V'',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. Supposing that V'' is distinct from V and V' by applying α-conversion if necessary,

$$D = \mathit{Derive}(V, V', e_t) = \mathit{Derive}(V, V' + V'', e_{t1}) \mathbin{@} \mathit{Derive}(V, V', e_{t2})$$

Since we can also assume that V'' is out of the reach of S, using the induction hypothesis we have:

$$\begin{array}{l} \mathit{Derive}(V, V' + V'', S(e_{t1})) = S(\mathit{Derive}(V, V' + V'', e_{t1})) \\ \mathit{Derive}(V, V', S(e_{t2})) = S(\mathit{Derive}(V, V', e_{t2})) \end{array}$$

Therefore $\mathit{Derive}(V, V', S(e_{t1})) = S(\mathit{Derive}(V, V' + V'', e_{t1})) \mathbin{@} \mathit{Derive}(V, V', S(e_{t2}))$.

The others The cases for rules with type generalization are proven same as the case of let bindings. For the other rules without constraint construction, the lemma is trivial.

□

Lemma 32 (Substitution lemma of constraint typing) *Let Γ be a typing environment, V_0 and V be distinct sets of type variables, e_t be a constraint annotated expression and S be a substitution where $Dom(S) = V$ and $\forall \nu \in V, DV(S(\nu)) = \emptyset$.*

Suppose e_t has a constraint typing $\Gamma, V_0 + V \Vdash e_t$ and

$$\{\tau_1 <: \mathcal{K}_1 \ \& \ldots \& \ \tau_n <: \mathcal{K}_n\} = \mathit{Derive}(V, V_0, e_t)$$

and for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. Then,

$$S(\Gamma), V_0 \Vdash S(e_t)$$

Proof: By induction over the construction of e_t:

Variables Let $e_t = \mathtt{x}_{\mathcal{K}} : \tau$. We have further two cases, depending on full or polymorphic instances:

Full instances If the typing is obtained by (kt-full-inst):

$$\frac{DV(\tau) \subset V + V_0 \quad \tau <: \mathcal{K} \quad \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) = \mathit{false}}{\Gamma, V + V_0 \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Because $S(\tau)$ replaces all the variable occurrences $\nu \in V$ to types without dynamic variables, $DV(S(\tau)) = DV(\tau) \setminus V \subset V_0$.

If $\mathit{generic}(\mathcal{K}) = \mathit{false}$, from the well formedness of Γ, $\mathcal{K}$ must be written simply as $[\sigma]$ for some type scheme σ. From the definition of *polyinst*, $\mathit{polyinst}(V, [\sigma], \tau) = \mathit{false}$. Since $\tau <: [\sigma]$, $\tau \leq \sigma$. From Lemma 4, $S(\tau) \leq S(\sigma)$, therefore $S(\tau) <: S([\sigma])$. Using (kt-full-inst), we have a typing for $S(e_t)$:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) <: S([\sigma]) \quad \mathit{polyinst}(V_0, S([\sigma]), S(\tau)) = \mathit{false}}{\Gamma, V_0 \Vdash \mathtt{x}_{S([\sigma])} : S(\tau)}$$

Otherwise, in the case $\mathit{generic}(\mathcal{K}) = \mathit{true}$, from

$$\begin{aligned} \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) &= (\mathit{generic}(\mathcal{K}) \ \& \ (DV(\tau) \cup DV(\mathcal{K})) \cap (V + V_0) \neq \emptyset) \\ &= \mathit{false} \end{aligned}$$

$DV(\tau) \cap (V + V_0) = \emptyset$ and $DV(\mathcal{K}) \cap (V + V_0) = \emptyset$. Therefore $DV(\tau) \cap V = \emptyset$ and $DV(\mathcal{K}) \cap V = \emptyset$, and since $Dom(S) = V$, $S(\tau) = \tau$ and $S(\mathcal{K}) = \mathcal{K}$. From these, $S(\mathtt{x}_{\mathcal{K}} : \tau)$ is also a full instance under V_0:

$$\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{polyinst}(V_0, \mathcal{K}, \tau) = \mathit{false}$$

The typing we want is like:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) = \tau <: \mathcal{K} = S(\mathcal{K}) \quad \mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Polymorphic instances The typing should be as follows:

$$\frac{DV(\tau) \subset V + V_0 \quad \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) = \mathit{true}}{\Gamma, V + V_0 \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Same as the case for the full instances, $DV(S(\tau)) \subset V_0$.

If $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{true}$, we can easily construct the goal by (kt-poly-inst):

$$\frac{DV(S(\tau)) \subset V_0 \quad \mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{true}}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Otherwise, if $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}$, Because $\mathit{polyinst}(V + V_0, \mathcal{K}, \tau) = \mathit{true}$,

$$(DV(\mathcal{K}) \cup DV(\tau)) \cap (V + V_0) \neq \emptyset$$

On the other hand, from $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}$, we have $(DV(S(\mathcal{K})) \cup DV(S(\tau))) \cap V_0 = \emptyset$. From the condition of S, this is rewritten as

$$(DV(\mathcal{K}) \cup DV(\tau)) \subset V$$

From these, we have $(DV(\mathcal{K}) \cup DV(\tau)) \cap V \neq \emptyset$, therefore $\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}$ and $\mathit{Derive}(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = \{\tau <: \mathcal{K}\}$. From the condition of $\mathit{Derive}(V, V', e_t)$, $S(\tau) <: S(\mathcal{K})$.

$S(\mathtt{x})$ becomes a valid full instance under V_0:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) <: S(\mathcal{K}) \quad \mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Let bindings Let $e_t = \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. We suppose V' is distinct with both V_0 and V, and out of the reach of S, so that we can avoid the same discussion about doubly bound type variables in proofs of the previous substitution lemmas. Then,

$$\frac{\begin{array}{c}\Gamma, V_0 + V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V_0 + V \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V_0 + V \Vdash \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

From the definition of *Derive*,

$$\begin{aligned}\mathit{Derive}(V, V_0, \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}) &= \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\} \\ &= \mathit{Derive}(V, V_0 + V', e_{t1})\ @\ \mathit{Derive}(V, V_0', e_{t2})\end{aligned}$$

Therefore there is an integer $1 \leq m \leq n$ and

$$\begin{array}{l}\mathit{Derive}(V, V_0 + V', e_{t1}) = \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_m <: \mathcal{K}_m\} \\ \mathit{Derive}(V, V_0', e_{t2}) = \{\tau_{m+1} <: \mathcal{K}_{m+1}\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\}\end{array}$$

and $S(\tau_i) <: S(\mathcal{K}_i)$ for all $1 \leq i \leq n$. We can apply the induction hypothesis to the premises of e_{t1} and e_{t2}, and obtain the followings:

$$\begin{array}{l}S(\Gamma), V_0 + V' \Vdash S(e_{t1}) \\ S(\Gamma\{\mathtt{x} : \mathcal{K}\}), V_0 \Vdash S(e_{t2})\end{array}$$

Now let's consider constraint constructions of $S(e_{t1})$. From Lemma 31, since V' is out of the reach of S, $\mathit{Derive}(V', \emptyset, S(e_{t1})) = S(\mathit{Derive}(V', \emptyset, e_{t1})) = S(D)$. From Lemma 28, $\mathit{adm}^+(\emptyset, S(\mathcal{K})) = \mathit{adm}^+(\emptyset, [\forall V'.S(\tau_1)\ S(D)]) = \mathit{true}$.

From all of these, using (kt-let), we prove:

$$\frac{\begin{array}{c}S(\Gamma), V_0 + V' \Vdash S(e_{t1} : \tau_1) \quad S(\Gamma)\{\mathtt{x} : S(\mathcal{K})\}, V_0 \Vdash S(e_{t2}) \\ S(D) = \mathit{Derive}(V', \emptyset, S(e_{t1})) \quad S(\mathcal{K}) = [\forall V'.S(\tau_1)\ S(D)] \quad \mathit{adm}^+(\emptyset, S(\mathcal{K})) = \mathit{true}\end{array}}{S(\Gamma), V_0 \Vdash S(\mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2})}$$

Recursive let bindings In this time, $e_t = \mathtt{let\ rec}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. Like what we have done in the part of normal lets, V' is distinct from V_0 and V and out of the reach of S.

The typing is:

$$\frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V_0 + V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_0 \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D] \quad \mu\kappa.\mathcal{K} \overset{\mu}{\to} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V_0 + V \Vdash \mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

Using the induction hypothesis we have the following:

$$\begin{array}{l}S(\Gamma)\{\mathtt{x} : S(\mu\kappa.\mathcal{K})\}, V_0 + V' \Vdash S(e_{t1} : \tau_1) \\ S(\Gamma)\{\mathtt{x} : S(\mu\kappa.\mathcal{K})\}, V_0 \Vdash S(e_{t2})\end{array}$$

By Lemma 31, $\mathit{Derive}(V', \emptyset, S(e_{t1})) = S(\mathit{Derive}(V', \emptyset, e_{t1})) = S(D)$ and $S(\mathcal{K}') = [\forall V'.S(\tau_1)\ S(D)]$. Since type substitutions and constraint substitutions are independent of each other,

$$S(\mu\kappa.\mathcal{K}) = \mu\kappa.S(\mathcal{K}) \overset{\mu}{\to} \{\kappa \Rightarrow \mu\kappa.S(\mathcal{K})\}(S(\mathcal{K})) = S(\{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}) = S(\mathcal{K}')$$

From Lemma 28, $\mathit{adm}^+(\emptyset, S(\mu\kappa.\mathcal{K})) = \mathit{true}$.

From these using (kt-letrec), we prove:

$$\frac{\begin{array}{c}S(\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), V_0 + V' \Vdash S(e_{t1}) \quad S(\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), V_0 \Vdash S(e_{t2}) \\ S(D) = \mathit{Derive}(V', \emptyset, S(e_{t1})) \quad S(\mathcal{K}') = [\forall V'.S(\tau_1)\ S(D)] \quad S(\mu\kappa.\mathcal{K}) \overset{\mu}{\to} S(\mathcal{K}') \\ \mathit{adm}^+(\emptyset, S(\mu\kappa.\mathcal{K})) = \mathit{true}\end{array}}{S(\Gamma), V_0 \Vdash S(\mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2})}$$

Generic bindings Let $e_t = \mathtt{generic\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t : \tau$. The typing is by (kt-generic), as follows:

$$\frac{\begin{array}{c}\Gamma, V + V_0 + V_i \Vdash e_{ti} : \tau_i \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, V + V_0 \Vdash e_t\end{array}}{\Gamma, V + V_0 \Vdash \mathtt{generic\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t}$$

Here we suppose for each σ_i, its bound type variables $BV(\sigma_i)$ are distinct from V and V_0.

Like the proof for let bindings, since the variable occurrences of e'_t are also occurrences of the whole expression e_t, their type annotation conditions for this lemma in e_t can be used as those for e'_t directly. Therefore from the induction hypothesis:

$$S(\Gamma\{\mathtt{x} : \mathcal{K}\}), V_0 \Vdash S(e'_t : \tau)$$

For each type case expression, we can prove that

$$\begin{array}{l}S(\Gamma), V_0 + V_i \Vdash S(e_{ti} : \tau_i) \\ S(D_i) = \mathit{Derive}(V_i, \emptyset, e_{ti})\end{array}$$

almost in the same way we have done in let bindings. From the closedness of σ_i, $S(\sigma_i) = \sigma_i$. From Lemma 28, $adm^+(\emptyset, S(\mathcal{K})) = true$

From these, using (kt-generic), we can prove the substituted version of the generic expression typing, using the closeness of σ_i:

$$\frac{\begin{array}{c} S(\Gamma), V_0 + V_i \Vdash S(e_{ti}) : \tau_i \quad S(D_i) = Derive(V_i, \emptyset, S(e_{ti})) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ S(\mathcal{K}) = [\sigma_1\ S(D_1) \mid \ldots \mid \sigma_n\ S(D_n)] \quad adm^+(\emptyset, S(\mathcal{K})) = true \\ S(\Gamma)\{\mathtt{x} : S(\mathcal{K})\}, V_0 \Vdash S(e_t) \end{array}}{\begin{array}{l} S(\Gamma), V_0 \Vdash \mathtt{generic}\ \mathtt{x}_{S(\mathcal{K})} = \mathtt{case} \\ \qquad\qquad \sigma_1 \Rightarrow S(e_{t1}) \mid \ldots \mid \sigma_n \Rightarrow S(e_{tn})\ \mathtt{in}\ S(e'_t) \end{array}}$$

Recursive generic bindings The case for recursive generic bindings is proven like normal generic bindings, combining the same argument in recursive lets.

The others The other expression constructions are non- generic and their typing are independent of generalized type variable sets. Therefore these are merely special cases of the normal substitution lemma of the typing, Theorem 3. We just prove the case of functions to show they are straightforward.

Let $e_t = \mathtt{fun}\ \mathtt{x} \rightarrow e'_t$. Its typing must be by (kt-fun):

$$\frac{DV(\tau_1) \subset V + V_0 \quad \Gamma\{\mathtt{x} : [\tau_1]\}, V + V_0 \Vdash e'_t}{\Gamma, V + V_0 \Vdash \mathtt{fun}\ \mathtt{x} \rightarrow e'_t}$$

All the variable occurrences $\mathtt{x}^i_{\mathcal{K}_i} : \tau_i$ in e'_t are also occurrences of e_t, therefore we can use the induction hypothesis for the constraint typing of e'_t and obtain:

$$S(\Gamma)\{\mathtt{x} : S([\tau_1])\}, V_0 \Vdash S(e'_t)$$

Like we have done in the part of variables, $DV(S([\tau_1])) \subset V_0$, because S maps all the type variables in V to types without dynamic type variables.

We can use (kt-fun) and put it back to a function:

$$\frac{DV(S(\tau_1)) \subset V_0 \quad S(\Gamma)\{\mathtt{x} : S([\tau_1])\}, V_0 \Vdash S(e'_t)}{S(\Gamma), V_0 \Vdash \mathtt{fun}\ \mathtt{x} \rightarrow S(e'_t)}$$

□

8.9.3 Well typed programs

Now the set of *well-typed programs* are restricted also by the generic constraint typing system: they are not only typable in the normal typing, but also must be checked by the constraint typing. More formally:

Definition 19 (Normal typing of constraint annotated expressions) *Let e_t be a full constraint annotated expression defined in Section 8.9.1 and Γ be a constraint typing environment. If $coerce(\Gamma) \vdash coerce(e_t)$, we simply write $\Gamma \vdash e_t$ and says e_t is well-typed in the normal typing under Γ.*

Definition 20 (well-typed programs) *A full constraint annotated expression e_t is a well-typed program if and only if $\emptyset \vdash e_t$ and $\emptyset, \emptyset \Vdash e_t$.*

8.10 Complete type soundness

Guarded by the normal and constraint typing systems, the well-typed programs are type sound: they are evaluated to neither of errors. First, the soundness from the normal typing assures that the well type programs never fails in *err*. This is easy because we have already it:

Theorem 31 (Non-generic type soundness) *If a full constraint annotated expression e_t has a normal typing in the empty typing environment, $\emptyset \vdash e_t$, and $\emptyset \models e_t$ terminates to some result r in the modified semantics, then r is not err.*

Proof: This is straightforward from Theorem 23. □

To complete the type soundness theorem, we need to work also with the constraint typing. First we define a new semantic typing with constraints, then prove the type soundness of the semantics using it.

Semantics with constraints

First of all, we modify semantic values and rules in Section 2.4 and Section 7.5.1. Expressions in closures and evaluation judgments are canonically extended with constraints. For example, in (e-gvar-r), the expression e_t and the variable $\mathtt{x}$ have the constraints in their annotations:

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

Semantic typing with constraints

We also extend the semantic typing so that it could know about the constraints. We replace (s-clos$*$), (s-gclos$*$), (s-gcase$*$) and (s-env) by rules with constraints, while the others are kept as they were (Figure 8.3). In the figure, typing environments with constraints Γ are coerced into one without constraints in the normal typing judgment $\Gamma \vdash e_t$ and the generalization function *Gen*. In the rules, we use natural translation from type schemes to constraints. Type scheme σ is regarded as a generic constraint $[\sigma\ \{\}]$ if required.

$$\textbf{(s-bcnst)}\ \frac{\tau \leq \sigma}{b_\sigma \in \tau} \qquad \textbf{(s-cnst)}\ \frac{\tau \leq_S \sigma}{S(cnst(c_\sigma)) \in \tau}$$

$$\textbf{(s-gen)}\ gen(\mathrm{g}_\sigma, \tau) \in \tau \qquad \textbf{(s-tpl)}\ \frac{v_1 \in \tau_1 \quad v_2 \in \tau_2}{v_1, v_2 \in \tau_1 * \tau_2}$$

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

$$\textbf{(s-kclos)}\ \frac{\begin{array}{c}\exists\Gamma,\ E \in \Gamma \\ \Gamma \vdash \mathtt{fun\ x} \to e_t : \tau_1 \to \tau_2 \\ \Gamma, \emptyset \Vdash \mathtt{fun\ x} \to e_t : \tau_1 \to \tau_2\end{array}}{clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \to \tau_2}$$

$$\textbf{(s-kclos-r)}\ \frac{\begin{array}{c}\exists\Gamma,\ E \in \Gamma \\ \Gamma\{\mathtt{f} : \forall V.\tau_1 \to \tau_2\} \vdash \mathtt{fun\ x} \to e_t : \tau_1 \to \tau_2 \\ \Gamma\{\mathtt{f} : \forall V.\tau_1 \to \tau_2\}, V \Vdash \mathtt{fun\ x} \to e_t : \tau_1 \to \tau_2 \\ \forall V.\tau_1 \to \tau_2 = Gen(\Gamma, \mathtt{fun\ x} \to e_t)\end{array}}{clos(E, \mathtt{f}, \mathtt{x}, e_t) \in \forall V.\tau_1 \to \tau_2}$$

$$\textbf{(s-kenv)}\ \frac{\forall (\mathtt{x} : \mathcal{K}) \in \Gamma,\ E(\mathtt{x}) = sv,\ sv \in \mathcal{K}}{E \in \Gamma}$$

$$\textbf{(s-kgclos)}\ \frac{\begin{array}{c}\exists\Gamma \quad E \in \Gamma \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad D = Derive(V, \emptyset, e_t) \\ \Gamma \vdash e_t \quad \Gamma, V \Vdash e_t\end{array}}{gclos(E, \_, V, e_t : \tau) \in [\forall V.\tau\ D]}$$

$$\textbf{(s-kgclos-r)}\ \frac{\begin{array}{c}\exists\Gamma \quad E \in \Gamma \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad D = Derive(V, \emptyset, e_t) \\ \mu\kappa.\mathcal{K} \stackrel{\mu}{\to} [\forall V.\tau\ D] \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_t \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{gclos(E, \mathtt{x}, V, e_t : \tau) \in \mu\kappa.\mathcal{K}}$$

$$\textbf{(s-kgcase)}\ \frac{\begin{array}{c}\exists\Gamma, E \in \Gamma, \\ \sigma_i = Gen(\Gamma, e_{ti}) \quad D_i = Derive(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma \vdash e_{ti} \quad \Gamma, BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K} = [\ \sigma_1\ D_1 \mid \ \ldots\ \mid \sigma_n\ D_n\ ]\end{array}}{gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mathcal{K}}$$

$$\textbf{(s-kgcase-r)}\ \frac{\begin{array}{c}\exists\Gamma, E \in \Gamma, \\ \sigma_i = Gen(\Gamma, e_{ti}) \quad D_i = Derive(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_{ti} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K}' = [\ \sigma_1\ D_1 \mid \ \ldots\ \mid \sigma_n\ D_n\ ] \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\to} \mathcal{K}'\end{array}}{gcase(E, \mathtt{x}, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mu\kappa.\mathcal{K}}$$

Figure 8.3: Semantic typing rules with constraints

The rules for closures (s-k$*$) check stored codes e_t in both normal and constraint typing systems. The same generalized type variables V must be used for the normal generalization and its corresponding constraint typing.

Theorem 32 **(Generic type soundness)** *If a fully type-annotated expression e_t has a typing $\Gamma \vdash e_t : \tau$ and $\Gamma, \emptyset \Vdash e_t$ under some constraint typing environment Γ, there exists E such that $E \in \Gamma$ and $E \models e_t : \tau \Longrightarrow r$, then r is neither err nor gerr but a value v such that $v \in \tau$.*

Proof: Proof by induction over the evaluation steps. The soundness of the non-generic part is already proven in Theorem 31, we just focus on generic stuffs:

Expressions related with non generic closures Although they have new conditions of constraint typings, proofs for their related cases are same as Theorem 31.

Variables of generic values Let $e_t : \tau = \mathtt{x}_{\mathcal{K}} : \tau$ and $\mathit{generic}(\mathcal{K}) = \mathit{true}$. Then from the semantic typing, $E(\mathtt{x})$ must be a generic closure, generic case closure, or their recursive versions:

Generic closures If $E(\mathtt{x}) = \mathit{gclos}(E', \_, V, e'_t : \tau')$, from the semantic typing of $E \in \Gamma$, $E(\mathtt{x}) \in \Gamma(\mathtt{x}) = \mathcal{K}$. By the semantic typing of the generic closure by (s-kgclos), $\mathcal{K}$ must have the form of $[\forall V.\tau'\ D]$ for some derived constraint list $D = \{\tau_i <: \mathcal{K}_i\}$:

$$\frac{\exists\Gamma', \quad E' \in \Gamma', \quad \forall V.\tau' = \mathit{Gen}(\Gamma', e'_t : \tau') \quad D = \{\tau_i <: \mathcal{K}_i\} = \mathit{Derive}(V, \emptyset, e'_t) \qquad \Gamma' \vdash e'_t \quad \Gamma', V \Vdash e'_t}{\mathit{gclos}(E', \_, V, e'_t : \tau') \in [\forall V.\tau'\ D]}$$

Since $\mathit{polyinst}(\emptyset, \mathcal{K}, \tau) = \mathit{false}$, $\mathtt{x}$ must be a full instance. The typings of $\mathtt{x}$ are by (t-inst) and (kt-full-inst) for some substitution S such that $\mathit{Dom}(S) = V$.

$$\frac{\Gamma(\mathtt{x}) = \mathit{coerce}(\mathcal{K}) = \forall V.\tau' \quad \tau \leq_S \forall V.\tau'}{\Gamma \vdash \mathtt{x}_{\forall V.\tau'} : \tau}$$

$$\frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset \emptyset \quad \mathit{polyinst}(\emptyset, \mathcal{K}, \tau) = \mathit{false} \quad \tau <: \mathcal{K}}{\Gamma, \emptyset \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Because $S(\tau') = \tau$ and $DV(\tau) = \emptyset$, $\forall \nu \in \mathit{Dom}(S) = V, DV(S(\nu)) = \emptyset$. From $\tau <: [\forall V.\tau'\ D]$, for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. Therefore, we can use Lemma 32 against the constraint typing of e'_t and obtain $S(\Gamma'), \emptyset \Vdash S(e'_t) : S(\tau') = \tau$. From the generalization, $FV(\Gamma') \cap V = \emptyset$ and $S(\Gamma') = \Gamma'$. Hence,

$$\Gamma', \emptyset \Vdash S(e'_t) : \tau$$

Also from the normal type substitution lemma Theorem 3,

$$\Gamma' \vdash S(e'_t) : \tau$$

The variable is evaluated by (e-gvar):

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

We can use the induction hypothesis for the evaluation $E' \models S(e'_t) \Longrightarrow r$, and obtain that r must be a value v such that $v \in \tau$.

Recursive generic closures In this case $E(\mathtt{x}) = gclos(E', \mathtt{y}, V, e'_t : \tau')$. The semantic typing $E \in \Gamma$ leads to $E(\mathtt{x}) \in \Gamma(\mathtt{x})$. By the semantic typing rule of the generic closure (s-kgclos-r), $\Gamma(\mathtt{x}) = \mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau'\ D]$ where

$$\begin{array}{l} \mu\kappa.[\forall V.\tau'\ D] \xrightarrow{\mu} [\forall V.\tau'\ D'] \\ D' = Derive(V, \emptyset, e'_t) \\ \forall V.\tau' = Gen(\Gamma, e'_t) \\ \Gamma\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \vdash e'_t \quad \Gamma\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \Vdash e'_t \end{array}$$

Since $polyinst(\emptyset, \mu\kappa.\mathcal{K}, \tau) = false$, $\mathtt{x}$ is a full instance. The typings of $\mathtt{x}$ are by (t-inst) and (kt-full-inst) for some substitution S such that $Dom(S) = V$.

$$\frac{\Gamma(\mathtt{x}) = coerce(\mu\kappa.\mathcal{K}) = \forall V.\tau' \quad \tau \leq_S \forall V.\tau'}{\Gamma \vdash \mathtt{x}_{\forall V.\tau'} : \tau}$$

$$\frac{\Gamma(\mathtt{x}) = \mu\kappa.\mathcal{K} \quad DV(\tau) \subset \emptyset \quad polyinst(\emptyset, \mu\kappa.\mathcal{K}, \tau) = false \quad \tau <: \mu\kappa.\mathcal{K}}{\Gamma, \emptyset \Vdash \mathtt{x}_{\mu\kappa.\mathcal{K}} : \tau}$$

Because $S(\tau') = \tau$ and $DV(\tau) = \emptyset$, $\forall \nu \in Dom(S) = V, DV(S(\nu)) = \emptyset$. From $\tau <: \mu\kappa.\mathcal{K}$, if we let $D' = \{\tau_i <: \mathcal{K}_i\}$, for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. We can use Lemma 32 against the constraint typing of e'_t and obtain $S(\Gamma'\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), \emptyset \Vdash S(e'_t) : S(\tau') = \tau$. From the generalization, $FV(\Gamma') \cap V = \emptyset$. Since $Dom(S) = V$, $S([\forall V.\tau'\ D']) = [\forall V.\tau'\ D']$. From this, we can easily prove that $S(\mu\kappa.\mathcal{K}) = \mu\kappa.\mathcal{K}$. Hence, $S(\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}) = \Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}$ and

$$\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}, \emptyset \Vdash S(e'_t) : \tau$$

Also from the normal type substitution lemma Theorem 3,

$$\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \vdash S(e'_t) : \tau$$

The variable is evaluated by (e-gvar-r):

$$\frac{\begin{array}{c}\exists S,\ \tau \leq_S \forall V.\tau' \\ E'\{\mathtt{y} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models S(e'_t : \tau') \Longrightarrow r\end{array}}{E\{\mathtt{x} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models \mathtt{x}_{\mu\kappa.\mathcal{K}} : \tau \Longrightarrow r}$$

Since $glos(E', \mathtt{y}, V, e'_t) \in \mu\kappa.\mathcal{K}$, we can use the induction hypothesis for the evaluation $E'\{\mathtt{y} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models S(e'_t) \Longrightarrow r$, and obtain that r must be a value v such that $v \in \tau$.

Generic case closures In the case $E(\mathtt{x}) = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1},\ \ldots,\ \sigma_n \Rightarrow e_{tn}])$, from $E(\mathtt{x}) \in \Gamma(\mathtt{x})$, if we let $\Gamma(\mathtt{x}) = \mathcal{K}$, the semantic typing of $E(\mathtt{x})$ is done by (s-kgcase) as follows:

$$\frac{\begin{array}{c}\exists\Gamma',\ E' \in \Gamma' \\ \sigma_i = \mathit{Gen}(\Gamma, e_{ti}) \quad D_i = \mathit{Derive}(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma' \vdash e_{ti} \quad \Gamma', BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K} = [\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]\end{array}}{\mathit{gcase}(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mathcal{K}}$$

Like in the case of generic closures, since $\mathit{polyinst}(\emptyset, \mathcal{K}, \tau) = \mathit{false}$, $\mathtt{x}$ must be a full instance. Therefore its constraint typing is done by (kt-full-inst):

$$\frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset \emptyset \quad \mathit{polyinst}(\emptyset, \mathcal{K}, \tau) = \mathit{false} \quad \tau <: \mathcal{K}}{\Gamma, \emptyset \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Because the constraint $\mathcal{K}$ is a pattern list $[\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]$ and $\tau <: \mathcal{K}$, by (k-fail-case), there must exist at least one type pattern which matches with τ: if we let the first such matched pattern be $\sigma_p\ d_p$, there exists a pattern substitution S whose domain is $BV(\sigma_p)$ and:

$$\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \quad \tau \leq_S \sigma_p \quad \tau <: \sigma_p\ D_p$$

If we let $D_p = \{\tau_j <: \mathcal{K}_j\}$, for all j, $S(\tau_j) <: S(\mathcal{K}_j)$.

Since $\tau \leq_S \sigma_p$ and $DV(\tau) = \emptyset$, $\forall \nu \in \mathit{Dom}(S), DV(S(\nu)) = \emptyset$ and we can use Lemma 32 and obtain $S(\Gamma'), \emptyset \Vdash S(e_{tp}) : \tau$. From the generalization of e_{tp}, we have $S(\Gamma') = \Gamma'$ and

$$\Gamma', \emptyset \Vdash S(e_{tp}) : \tau$$

We obtain $\Gamma' \vdash S(e_{tp})$ from the normal typing $\Gamma' \vdash e_{tp}$ using Theorem 3.

The evaluation of $\mathtt{x}_{\mathcal{K}} : \tau$ under E is done by (e-gcase) as follows:

$$\frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \quad \tau \leq_S \sigma_p \\ E' \models S(e_{tp}) \Longrightarrow r\end{array}}{E\{\mathtt{x} = \mathit{gcase}(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x} : \tau \Longrightarrow r}$$

From all the above we can use the induction hypothesis and get $E' \models S(e_{tp}) \Longrightarrow v$ for some value v.

Recursive generic case closures Provable as same as the non recursive generic case closures, using also the discussions we have done in the case of the recursive generic closures.

Generic let $e_t = \mathtt{let}\ \mathtt{x}_{V,\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}$. Because $\Gamma \vdash e_t$ and $\Gamma, \emptyset \Vdash e_t$, the typings must be done by (t-let) and (kt-let):

$$\frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau\end{array}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2}} \qquad \frac{\begin{array}{c}\Gamma, V \Vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e_{t2} \\ \mathcal{K} = [\forall V.\tau_1\ D] \\ D = \mathit{Derive}(V, \emptyset, e_{t1}) \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, \emptyset \Vdash \mathtt{let}\ \mathtt{x}_{V,\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}}$$

The evaluation is done by (e-g-let):

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

Using (s-kgclos), the typing of the generic closure $gclos(E, \_, V, e_{t1})$ is:

$$\frac{\begin{array}{c} E \in \Gamma \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1}) \quad D = Derive(V, \emptyset, e_{t1}) \\ \Gamma \vdash e_{t1} \quad \Gamma, V \Vdash e_{t1} \end{array}}{gclos(E, \_, V, e_{t1}) \in [\forall V.\tau_1\ D] = \mathcal{K}}$$

It leads to $E\{\mathtt{x} = gclos(E, \_, V, e_{t1})\} \in \Gamma\{\mathtt{x} : \mathcal{K}\}$. We can apply the induction hypothesis to the evaluation of e_{t2}: r must be a value $v \in \tau$.

Generic recursive let This time, $e_t = \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2}$. The attached constraint to $\mathtt{x}$ must be a recursive constraint $\mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau_1\ D]$ from the following typings by (t-letrec) and (kt-letrec):

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

$$\frac{\begin{array}{c} \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, \emptyset \Vdash e_{t2} \\ D' = Derive(V, \emptyset, e_{t1}) \quad \mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau_1\ D] \stackrel{\mu}{\to} [\forall V.\tau\ D'] \\ adm^{+}(\emptyset, \mu\kappa.\mathcal{K}) = true \end{array}}{\Gamma, \emptyset \Vdash \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2}}$$

From the normal typing of e'_{t1} and its generalization, $\Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t1} : \tau_1$ by Lemma 7.

The evaluation is done by (e-g-letrec):

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \mathtt{x}, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

From the above typings by (s-kgclos-r),

$$\frac{\begin{array}{c} E \in \Gamma \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1}) \quad D' = Derive(V, \emptyset, e_{t1}) \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\to} [\forall V.\tau_1\ D'] \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t1} \end{array}}{gclos(E, \mathtt{x}, V, e_{t1}) \in \mu\kappa.\mathcal{K}}$$

Therefore $E\{\mathtt{x} = gclos(E, \mathtt{x}, V, e_{t1})\} \in \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}$. We can use the induction hypothesis for the evaluation of e_{t2}. r is a value v where $v \in \tau$.

Generic bindings This is the case $e_t = \mathtt{generic\ x}_{\mathcal{K}}\,\mathtt{=}\,\mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t$.

The normal typing is by (t-generic):

$$\frac{\begin{array}{c}\Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \\ \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e'_t : \tau\end{array}}{\Gamma \vdash \mathtt{generic}\ \mathtt{x}_\sigma\ \mathtt{=}\ \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t : \tau}$$

and (kt-generic) for the constraint typing:

$$\frac{\begin{array}{c}\Gamma, V_i \Vdash e_{ti} : \tau_i \quad D_i = Derive(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad adm^+(\emptyset, \mathcal{K}) = true \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e'_t\end{array}}{\Gamma, \emptyset \Vdash \mathtt{generic}\ \mathtt{x}_{\sigma\mathcal{K}}\ \mathtt{=}\ \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t}$$

The expression is evaluated by (e-generic):

$$\frac{E\{\mathtt{x} = gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models e'_t \Longrightarrow r}{E \models \mathtt{generic}\ \mathtt{x}_{\mathcal{K}}\ \mathtt{=}\ \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t \Longrightarrow r}$$

The generic case closure created by this evaluation can be typed by (s-kgcase), using premises obtained from the typings:

$$\frac{\begin{array}{c}E \in \Gamma, \\ \sigma_i = Gen(\Gamma, e_{ti}) \quad D_i = Derive(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma \vdash e_{ti} \quad \Gamma, BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K} = [\ \sigma_1\ D_1 \mid \ \ldots \ \mid \sigma_n\ D_n\ ]\end{array}}{gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mathcal{K}}$$

We can use the induction hypothesis for the evaluation of e'_t and it must be successful, since from the semantic typing, $E\{\mathtt{x} = gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \in \Gamma\{\mathtt{x} : \mathcal{K}\}$ and the normal and constraint typings give us $\Gamma\{\mathtt{x} : \mathcal{K}\} \vdash e'_t : \tau$ and $\Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e'_t$.

Recursive generic bindings This case is proven in the same way as the case of non recursive generic bindings. The rules in the proof must be replaced by their recursive counterparts.

□

This theorem is immediately simplified to the following statement:

Theorem 33 (Generic type soundness of well-typed programs) *Let e_t be a full constraint annotated, well-typed program. Then evaluation $\emptyset \models e_t$ returns neither err nor gerr, but always some value v.*

Proof: This is a special case of Theorem 32, when $\Gamma = \emptyset$ and $E = \emptyset$. □

8.11 Dynamic variable attribute in constraints

The distinction of regular and dynamic type variables has played essential role in every aspect of extensional polymorphism: semantics, typing and compilation. With the help of generic constraints, however, these two kinds of type variables can be merged into one. This merge simplifies the type algebra, identical to the one of the normal ML. With small assumption and modification to generic constraints, "dynamic" attribute of type variables can be deduced by examining constraints. We here outline this idea.

Pattern constraints

If we suppose that user-defined generic values always have more than one type cases, user-defined generic values and the other values can be distinguished by checking whether their constraints contain only one type pattern constraint or more. For example, a normal polymorphic value `List.map` has the following constraint of singleton type pattern:

$$\texttt{List.map : } [\forall\alpha\beta\texttt{.}(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}]$$

identical to its type scheme in the normal typing. On the other hand, a use generic value `plus` which is overridden by addition functions for integers and floats have the following constraint with two type patterns:

$$\mathcal{K}_{\texttt{plus}} \ = [\texttt{int} \to \texttt{int} \to \texttt{int} \ | \ \texttt{float} \to \texttt{float} \to \texttt{float}]$$

Its type scheme in the normal typing is identical to the least general anti-unifier of these type patterns, and therefore in the new simplified type algebra, it is $\forall\alpha\texttt{.}\alpha \to \alpha \to \alpha$. Though it has no longer dynamic attribute, looking at its generic constraint with two patterns, it is clear that `plus` is a generic value and the type scheme must be dynamic, which is $\forall\delta\texttt{.}\delta \to \delta \to \delta$ in the current type algebra.

Derived constraints

Derived generic values have only one element of type pattern, same as the normal values. But their dynamic type scheme can be deduced, since they have derived constraints. For example, the following derived generic value `double`

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

has a constraint $[\forall\delta\texttt{.}\delta \to \delta\ \{\delta \to \delta\ <: \mathcal{K}_{\texttt{plus}}\}]$ in the current type system. In the new simplified type algebra, we can remove the explicit dynamic attribute of δ, since it is clear from the existence of the derived constraint:

$$[\forall\alpha\texttt{.}\alpha \to \alpha\ \{\alpha \to \alpha\ <: \mathcal{K}_{\texttt{plus}}\}]$$

and the type scheme of `double` can be written as $\forall\alpha\texttt{.}\alpha \to \alpha$.

Generic primitives

Since generic primitives are defined as totally polymorphic, their type schemes are singleton lists of their type schemes, $[\sigma]$. Here we cannot just remove the dynamic attributes from σ, since such "undynamized" constraints of singleton type pattern without derived constraints could not distinguished from constraints of non-generic values. Therefore we introduce a new form of derived constraint $\texttt{generic}(\alpha)$ to indicate their genericity explicitly. For example, a generic primitive `print` whose type scheme is $\forall\delta.\delta \rightarrow \texttt{unit}$ in the current type system will have the following generic constraint in the new simplified type algebra:

$$[\forall\alpha.\alpha \rightarrow \texttt{unit}\ \{\texttt{generic}(\alpha)\}]$$

Thanks to this new constraint, `print` is identified as a generic primitive.

Dynamic attribute is still useful internally

Thus, using generic constraints, the distinction between regular and dynamic type variables is no longer necessary. This will lower the barrier to use extensional polymorphism a lot, since users need not learn their difference. At a glance, the normal typing layer of extensional polymorphism is now equivalent to the classical ML typing, except the newly added `generic` bindings.

In spite of this simplification of type algebra, the original dynamic attribute of type variables is still helpful for the constraint typing and compilation, since they handle generic values differently from non-generic values. Therefore in the following chapters, we will continue using this distinction of type variables.

8.12 Related works

8.12.1 Abstract interpretation

In the initial study of extensional polymorphism by Dubois, Rouaix and Weis [DRW95], static detection of type failures of generic instances is not by types, but based on abstract interpretation of programs. The abstraction of a program is a predicate on types that captures type abstractions, type matching, and type applications extracted from its compilation. For example:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
in
let double = fun x → plus x x in
double 1;;
```

```
let abs_plus t = match t with
| «int» → true
| «float» → true
| _ → false (* type match failure *)
in
let abs_double =
  fun t → abs_plus t in
abs_double «int»;;
```

The left hand program has the abstraction in the right. Evaluating this abstraction and simulates all the possible type applications and matchings, the type safety of generic instances is verified.

Abstractions are not types and are too abstract, therefore type system cannot make use of them for overloading resolution. As a result, type schemes of [DRW95] are always full parametric polymorphic, which are same as in our normal typing layer. The lack of non-exhaustiveness of generic values in type system prevents overload resolution of generic instances: users must often manually constrain instance types of generic values, even when overloading resolutions are trivial from definitions.

Our generic constraints can be regarded as a translation of these abstraction interpretations in type algebra. Having non-exhaustive information of generic values, type reconstruction now can resolve their instance types.

8.13 Conclusion and discussion

In this chapter we constructed generic constraint type system which statically detects all the invalid generic instances which could raise generic run time type errors. Combining this and the normal typing, we can make sure that well-typed GML programs cannot be evaluated to any kind of errors.

Two-layer design

We formalized the constraint typing as an independent system of the normal typing which is based on the classical ML typing. This separation benefits both theory and implementation. As we have seen, to detect illegal user-defined generic instance types which cannot match with any type pattern, type information of values in the constraint typing is inevitable to become much more complicated than normal type schemes. Migrating such type information directly into the normal and classical type system would extend it to something too hard to understand. Thanks to the two-layer design of type systems, the first layer of the normal typing is kept as a mild extension of ML typing system. Also in implementation, we can implement type reconstruction into two steps: first we type expressions by the normal type reconstruction algorithm in Section 2.6. This is rather easily achieved by modifying some existing ML implementation to add dynamic type variables and related rules. Then type-annotated expressions by this first reconstruction are sent to the second reconstruction phase which detects invalid generic instances. Since code for the first phase is isolated from the second, maintenance of the first phase can be done even by who do not understand the generic constraint typing at all.

Generic constraints

In the generic constraint typing system, a value is associated with a generic constraint, type information which denotes the set of the types where semantics of the value is defined. A constraint is basically represented as a list of type pattern constraints, type scheme where the corresponding value is well defined. Pattern constraints may also contain derived constraints, the information of internal polymorphic generic instances whose validity is dependent on full instantiation of the whole constraint. Constraints may also be abstracted by constraint variables to express recursiveness. Due to this recursively defined constraints, valid generic instance type relationship $\tau <: \mathcal{K}$ is undecidable, unless we introduce some restriction to the recursive constraints. In order to make the instance relationship decidable, we introduced admissible constraint restriction which limits constraints so that size of run time dispatching of generic values must decrease monotonically (but need not to be strictly) in the course of recursive calls of GML compilation. Unfortunately, we have some useful but inadmissible generic values which are therefore impossible to define, even though their instance check always terminates. In actual implementation, however, this admissibility restriction will not be necessary, since in type reconstruction of generic instances (Chapter 10) we use another restriction which limits maximum number of reconstruction algorithm recursive calls. Under this restriction, type checking is assured to always terminate even with non-admissible constraints. However, since non-admissible generic definitions tend to be meaningless, our type checker should print out warnings when it finds them.

Typing rules

Typing rules were defined using generic constraints. Under an assumption that expressions are already well-typed by the normal typing, these rules only check generic related constructions. To prove the generic type soundness, we extended the semantic typing system so that they can be also aware of constraint typing, additional to the normal typing. In some sense this is the only part where both typing layers meet together; the soundness proof uses typings of both layers intensively.

No free dynamic variables

The free occurrences of dynamic type variables are restricted to the generalized variable set of typing judgment: Lemma 29 says that if $\Gamma, V \Vdash e_t$ holds, $DV(e_t) \subset V$. This means that in a well-typed program e_t whose typing is $\emptyset, \emptyset \Vdash e_t$, dynamic type variables cannot occur freely: they must be generalized by some polymorphism. This restriction might be too strong. For example, in the following program,

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| string → string → string ⇒ (^)
| ∀δ.δ list → δ list → δ list ⇒ (@)
in
plus [] [];;
```

the instance `plus` cannot have a type δ `list` $\rightarrow \delta$ `list` $\rightarrow \delta$ `list`, even though the expression can be evaluated to a null list without any problem with this type annotation.

It is highly possible to remove the restriction of free dynamic type variable occurrences from the generic constraint type system without big difficulty, but it is a future issue.