Type class elaboration

Final exam, MPRI 2-4

2022/03/09 — Duration: 2h45

Although the questions are in English, it is permitted to answer in either French or English—and recommended to answer in French if this is your native language.

The topic of this exam is *type classes*.

In Sections 1 and 2, we ask you to *implement* an algorithm discussed in Didier Rémy's lecture notes for this course, namely elaboration of type-class dictionaries by instance search.

In Section 3, we use an example to discuss coherence – the property, that may or may not hold of a specific type-class system, that all ways to elaborate a type-class constraint results in the same runtime behavior.

In Section 4, we discuss an extension to type classes, *associated types*, that require a more advanced type system for elaborated programs (System Fω).

Note. *For implementation-related questions, we specify the questions using OCaml. We recommend using OCaml to answer it if you are comfortable enough with the language (it makes our life simpler), but you can use a different programming language if you want to. In this case, it is your responsibility to ensure that your adaptation of the provided prototypes/types/specification is correct – that you are not answering a different question. If the mapping is non-obvious, please explain why you chose it that way.*

Note. *For implementation-related questions, your code will be checked by humans, not machines. You will not get penalized for getting minor syntactic details wrong, as long as we agree that the code is morally correct.*

1 An effect for search problems

The search algorithm in Section 2 will either fail, or return a single solution. As a warm-up, you will define a simple applicative functor that captures "looking for the first solution of a search problem", to use in your code in Section 2.

Note. *In a real program we may want to express "lazily enumerate all the solutions of a search problem", and then only take the first solution. But our less general setting makes for an easier exam!*

We define a type

```
type α search = unit → α option
```

of computations that either return a value of type α or fail with no value. (The `(unit → ...)` thunking ensures that search is not performed right away, but only when the computation is explicitly `run`; it would not be necessary in a lazy language.)

Reminder: `option` is defined as follows in the standard library:

```
type α option =
  | None
  | Some of α
```

Question 1. *Define the following functions:*

```
val return : α → α search
val fail : α search

val run : α search → α option
```

Answer

□

Question 2. *Define the following operators for the "applicative functor" and "monad" structure:*

```
val map : (α → β) → α search → β search
val pair : α search → β search → (α * β) search
val bind : α search → (α → β search) → β search
```

Answer

□

You may use the following binding operators as syntactic sugar in your code:

```
let ( let+ ) s f = map f s
let ( and+ ) = pair
let ( let* ) = bind
```

Reminder: `(let+ x = e1 and+ y = e2 in foo)` desugars into

```
( let+ )
  (( and+ ) e1 e2)
  (fun (x, y) → foo)
```

Question 3. *Define* `sum` *and* `traverse` *that take several computations and return the result of the first computation that succeeds. (Trying computations in a left-to-right order is more natural, but not a strict requirement.)*

```
val sum : α search → α search → α search
val traverse : α search list → α list search
```

We also define an infix operator

```
let ( ++ ) = sum
```

so you can write `a ++ b ++ c` *for* `sum a (sum b c)`*.*

If both `a` *and* `b` *succeed, the computational cost of running* `sum a b` *should be* `max(cost(a), cost(b))` *in the worst case, not* `cost(a)+cost(b)`*.*

Answer

□

Question 4. *Show, by equational reasoning, that:*

- `fail` *is an absorbing element for* `(let+)`*:* `(let+ x = fail in t)`*, which unfolds into* `(map (fun x → t) fail)` *is equivalent to* `fail`*,*
- `fail` *is a neutral element for* `++`/`sum`*:* `(fail ++ t)` *and* `(t ++ fail)` *are equivalent to* `t`

By "`t` *is equivalent to* `u`*" here, we mean that using* `run` *on the two expressions returns identical results. We expect a proof by equational reasoning, proceeding as a series rewrite steps from one expression to the other. (More precisely, β-η equivalences for functions and datatypes suffice in our examples.)*

For example, here is a very simple proof in this style that `Option.map (fun x → x + 1) None` *is* `None`*:*

```
  Option.map (fun x → x + 1) None
=
  (fun f o → match o with
    | None → None
    | Some v → Some (f v)
  ) (fun x → x + 1) None
=
  match None with
  | None → None
  | Some v → Some ((fun x → x + 1) v)
=
  None
```

Answer

□

2 Searching for type class elaboration

In this section, we are going to implement type class elaboration following Didier Remy's course notes, for a small Haskell fragment whose surface syntax looks like this:

```
class Eq α = ...
class (u : Eq α) ⇒ Ord α = ...
instance i : Ord Int = ...

sort : ∀α. (z : Ord α) ⇒ List α → List α
sort = ...
```

In the example above, we bind a variable to the subclass hypothesis `(u : Eq α) ⇒ ...` of the class `Ord α`, and also in the type class constraint `(z : Ord α) ⇒ ...` of the function `sort`. There are no such variables in real Haskell programs, as the user code does not need to name the corresponding projections or instances. But we use them to be able to write the elaborated dictionary terms.

For example, if the implementation of `sort` was to somehow involve the constraint `Eq α`, this contraint could be elaborated into the dictionary term `u α z`, as we can see above that `u : forall α. Ord α → Eq α` is the projection coming from the class definition of `Ord α`, and `z` is the variable representing the `Ord α` instance.

The relevant course notes of Didier Rémy are included at the very end of this document. They include more details than you need for this exam, we are not encouraging you to read them in full. Here we will just summarize the key syntactic categories and the judgment that you will be implementing as a search procedure:

G	type constructor names	$q ::=$	dictionary terms
K	class names	$\mid u$	projection variables
$P ::= \mathsf{K}\ \alpha$	class constraints on type variables	$\mid z$	local assumption variables
$Q ::= \mathsf{K}\ \tau$	class constraints on types	$\mid q\ \tau$	type applications
		$\mid q\ q$	applications

Figure 7.4, page 158: Algorithmic typing rules for dictionary terms:

D-Ovar-Inst
$$\frac{z : \forall\vec{\beta}.P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma \qquad \forall i \in 1..n,\ \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}$$

D-Proj
$$\frac{u : \forall\alpha.\ \mathsf{K}'\ \alpha \Rightarrow \mathsf{K}\ \alpha \in \Gamma \qquad \Gamma \vdash q : \mathsf{K}'\ \tau}{\Gamma \vdash u\ \tau\ q : \mathsf{K}\ \tau}$$

D-Var
$$\frac{z : \mathsf{K}\ \alpha \in \Gamma}{\Gamma \vdash z : \mathsf{K}\ \alpha}$$

Note. *The rest of this Section goes as follows: a lot of type definitions for the grammatical categories involved (types, dictionary terms, environments...), then one big programming question. In the middle of the "type definitions" part there are many small "warmup questions" to define very simple examples using the type definitions, to help gradual understanding.*

Types

We propose the following type definitions (in the implementation) for types (in the object language).

```
type constr_name = Constr of string (* G *)
type tyvar = TyVar of string (* α *)

type typ (* τ *) =
  | TyVar of tyvar (* α *)
  | TyConstr of constr_name * typ list (* G τ̄ *)

type shallow_typ = constr_name * tyvar list (* G β̄ *)
```

Question 5. *Define a variable* `int` *of type* `shallow_typ` *that represents the type* `Int`*:*

```
let int : shallow_typ = ...
```

We are asking you to name this expression as you may want to reuse it in answers to following questions.

Answer

□

Class names and class expressions

```
type class_name = Class of string (* K *)

(* P ::= K α
   K (G β̄)      (no name given by Didier)
   Q ::= K τ *)
type α class_expr = class_name * α
```

Question 6. *Define a variable* `ord_alpha` *of type* `tyvar class_expr` *that represents* `Ord` α*.*
Define a variable `eq_int` *of type* `shallow_typ class_expr` *that represents* `Eq Int`*.*

Answer

□

Term variables and dictionary terms

We define types `tevar` of term variables (note: term variables represent both projection functions u from a class to a subclass, and instance variables h to refer to specific instances and build bigger instance terms), and a type `term` representing dictionary terms q as classified by the "algorithmic typing rules" of the course (Figure 7.4 page 158):

```
(* x, u, z *)
type tevar = TeVar of string

(* Γ ⊢ q : Q *)
type term =
  | Inst of tevar * typ list * term list (* z τ̄ q̄ *)
  | Proj of tevar * typ * term (* u τ q *)
  | Var of tevar (* z *)
```

Question 7. *In the declaration environment*

```
class Eq α = ...
class (u : Eq α) ⇒ Ord α = ...
instance i : Ord Int = ...
```

Write the representation of the dictionary term $(u\ \mathsf{Int}\ i)$ *for* `Eq Int`*, as a value of type* `term`*.*

Answer

□

Class instances

We define a type `class_instance` of instance declarations,

```
(* instance h : ∀β̄. P̄ ⇒ K(G β̄) *)
type class_instance = {
  params: tyvar list; (* β̄ *)
  dependencies: tyvar class_expr list; (* P̄ *)
  class_name: class_name; (* K *)
  constr_name: constr_name; (* G *)
}
```

Question 8. *Define* `inst_ord_int` *of type* `class_instance`*, representing the declaration* `instance z : Ord Int`*.*

Answer

□

Elaboration environments

Finally, we define a type `env` of typing environments in which dictionary terms are elaborated: free term variables corresponding to projections of a class into its subclass, free term variables corresponding to instance declarations, free type variables from the user program, and free type variables coming from instance assumptions in the user code.

```
type env = {
  projections: (tevar * class_name * class_name) list; (* u : ∀α. K α ⇒ K' α *)
  instances: (tevar * class_instance) list;
  tyvars: tyvar list;
  vars: (tevar * tyvar class_expr) list;
}
```

Question 9. *Give the representation of the elaboration environment for the body of* `sort` *as in our running example:*

```
class Eq α = ...
class (u : Eq α) ⇒ Ord α = ...
instance i : Ord Int = ...

sort : ∀α. (z : Ord α) ⇒ List α → List α
sort = ...
```

(This environment contains the classes `Eq` *and* `Ord`*, an instance* `i : Ord Int`*, a type variable α and a local dictionary variable* `z : Ord α`*.)*

Answer

□

Main question

Question 10. *Implement*

```
val elaborate : env → typ class_expr → term search
```

You are of course free to define auxiliary functions, etc. (If it makes your life easier, it's fine if the auxiliary function definitions come after their usage in the code. A human reader will be able to reorder your literate program.)

As an indication of what to expect, our solution fits in less than 100 lines of code, including a couple (well-chosen) auxiliary functions.

Answer

□

3 Dictionary implementation and coherence

In Section 2, our search procedure returns elaborated terms where class projections and class instances are represented as free variables (u, z...). The course notes explain how to give concrete definitions to these free variables such that elaboration produces terms that *compute* dictionary values.

For example, the class definitions

```
class Eq α { equal: α → α → Bool }
class (u : Eq α) ⇒ Ord α { less : α → α → Bool }
```

get elaborated into the following record types

```
type α eq = { equal : α → α → bool }
type α ord = { eq : α eq; less : α → α → bool }
```

and we can define the projection `u` as the simple function

```
let u ord_inst = ord_inst.eq
```

Defining this realization of types, instances and projections in full generality is a bit tedious, as seen in the course notes. We ask you to do it again in simple examples, to discuss questions related to coherence.

Question 11. *Give two different dictionary terms that are correct elaborations for the constraint* `Eq (Int * String)` *in the following environments:*

```
class Eq α = { ... }
instance eq_int : Eq int
instance eq_string : Eq string
instance eq_prod : Eq α, Eq β ⇒ Eq (α * β)

class (eq_of_ord : Eq α) ⇒ Ord α = { ... }
instance ord_int : Ord int
instance ord_string : Ord string
instance ord_prod : Ord α, Ord β ⇒ Ord (α * β)
```

Answer

□

Question 12. *We assume that the following definitions of combinators for equality and comparison of integers, strings and pairs are given:*

```
equal_int : int → int → bool
equal_string : string → string → bool
equal_prod : (α → α → bool) → (β → β → bool) → α * β → α * b → bool

less_int : int → int → bool
less_string : string → string → bool
less_prod : (α → α → bool) → (β → β → bool) → α * β → α * b → bool
```

(We use abbreviated words "eq" and "ord" to refer to the classes, and full words "equal" and "less" for their operations.)

Give term definitions that realize the following instances from our environment:

```
instance eq_int : Eq int
instance eq_prod : Eq α, Eq β ⇒ Eq (α * β)

instance ord_int : Ord int
instance ord_prod : Ord α, Ord β ⇒ Ord (α * β)
```

For example, for `ord_prod` *you are expected to implement*

```
val ord_prod : α ord → β ord → (α * β) ord
```

Answer

□

Question 13. *Show, by equational reasoning, that the two dictionary terms of Question 11 have equivalent meanings when their free variables are bound to your definitions in Question 12.*

Answer

□

4 Associated types

In this section we explore an extension of our language of type classes (MiniHaskell) with "associated types", which associate a type to each instance of a type class.

Our working example is the following:

```
class Set α = {
  type repr;
  empty : repr;
  add : α → repr → repr;
  mem : α → repr → bool;
}
```

The class `Set` α represents an implementation of sets of values of type α. Different instances of this class can choose different concrete representations for sets (for example: sets of booleans can be represented compactly as bitsets, sets of integers also enjoy efficient encodings...) by defining a type `repr`, the "representation" of sets, that is "associated" to this instance. The methods of the class `empty`, `add` and `mem` depend on this type `repr`.

```
instance set_bool : Set bool = {
  type repr = bitset;
  empty = ...;
  add = ...;
  mem = ...;
}
```

(We assume given a type `bitset` of bit sets.)

Values that are parametrized over an instance `z : Set foo` can mention the type `z.repr` associated with the instance `z`:

```
val mem_both : (set : Set α) ⇒ α → α → set.repr → bool
let mem_both v1 v2 s = (mem v1 s && mem v2 s)
```

We propose to elaborate the type of the class declaration `Set` α above by dictionaries using the following record type:

```
type (α, 'repr) set = {
  empty : 'repr;
  add : α → 'repr → 'repr;
  mem : α → 'repr → bool;
}
```

Your task in the rest of this section is to figure out the *rest* of the elaboration, and scale it to the case of *parametrized* associated types.

Question 14. *Give the type of the elaboration of* `instance set_bool` *above:*

```
val set_bool : ...
```

Answer

□

Question 15. *Give the type of the elaboration of the function below?*

```
val set_of_list : (set : Set α) ⇒ α list → set.repr
```

Answer

□

Let us consider a type class for associative maps, rather than sets, of values of type α.

```
class Map α = {
  type β repr;
  empty : β repr;
  add : α → β repr → β repr;
  find : α → β repr → β option;
}
```

We propose the following elaboration in an imaginary extension of OCaml/ML with Fω type abstractions:

```
type (α, 'repr) map = {
  empty : β. β 'repr;
  add : β. α → β → β 'repr → β 'repr;
  find : β. α → β 'repr → α option
}
```

More precisely:

- β `'repr` is imaginary OCaml syntax for applying the type parameter `'repr`, of higher kind `* → *`, to the type parameter β. (Yes, it would be less confusing if OCaml wrote its type-level function applications with the function first, or if everyone else followed the rightful convention of function application with the argument first.)

- `(`β `.` $\beta \to \beta$`)` is (real) OCaml syntax for polymorphic record fields, it can be read as `(forall` β`. (`$\beta \to \beta$`))`.

Question 16. *In this imaginary extension of OCaml (or your language of choice) with* Fω *types, give the type of the elaborations of*

```
instance int_map : Int Map = {
  type β repr = β patricia_trie
  ...
}

val map_of_list : (map : Map α) ⇒ (α * β) list → β map.repr
```

Answer

□

Cours notes from Didier Rémy

What follows is a verbatim citation of the course notes of Didier Rémy that are relevant to this exam.

(1995) but did not take over, because of their restrictions. Recent works on type classes is still going on Morris and Jones (2010).

In the rest of this chapter we introduce a tiny language called Mini Haskell that models the essence of Haskell type classes; at the end we also discuss *implicit arguments* as a less structured but simpler way of introducing dynamic overloading in a programming language.

7.2 Mini Haskell

Mini Haskell—or MH for short—is a simplification of Haskell to avoid most of the difficulties of type classes but keeping their essence: it is restricted to single parameter type classes and no overlapping instance definitions; it is close in expressiveness and simplicity to *A second look at overloading* by Odersky et al. but closer to Haskell in style—it can be easily generalized by lifting restrictions without changing the framework.

The language MH is explicitly typed. In this section, we first present some examples in MH, and then describe the language and its elaboration into System F. We introduce an implicitly-typed version of MH and its elaboration in the next section.

7.2.1 Examples in MH

An equality class and several instances many be defined in Mini Haskell as follows:

```
class Eq (X)    { equal : X → X → Bool }
inst  Eq (Int)  { equal = primEqInt }
inst  Eq (Char) { equal = primEqChar }
inst  Λ(X) Eq (X) ⇒ Eq (List (X))
  { equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
            | [],[]  → true | [], _ | _,[]  → false
            | h1::t1, h2::t2 → equal X h1 h2 && equal (List X) t1 t2 }
```

This code declares a class (dictionary) of type Eq(X) that contains definitions for $\mathrm{equal} : X \to X \to \mathit{Bool}$ and creates two concrete instances (dictionaries) of type $\mathit{Eq}(\mathit{Int})$ and $\mathit{Eq}(\mathit{Char})$, and a function that, given a dictionary for $\mathit{Eq}(X)$, builds a dictionary for type $\mathit{List}(X)$. This code can be elaborated by explicitly building dictionaries as records of functions:

```
type Eq (X) = { equal : X → X → Bool }
let equal X (EqX : Eq X) : X → X → Bool = EqX.equal

let EqInt : Eq Int   = { equal = ( primEqInt : Int → Int → Bool ) }
let EqChar : Eq Char = { equal = primEqChar }

let EqList X (EqX : Eq X) : Eq (List X) =
  { equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
      | [],[]  → true | [], _ | [], _ → false
      | h1::t1, h2::t2 →
```

$$\text{equal}\ X\ \text{EqX}\ h_1\ h_2\ \&\&\ \text{equal}\ (\mathit{List}\ X)\ (\text{EqList}\ X\ \text{EqX})\ t_1\ t_2\ \}$$

Classes may themselves depend on other classes (called superclasses), which realizes a form of class inheritance.

```
class $\mathit{Eq}\ (X) \Rightarrow \mathit{Ord}\ (X)\ \{\ \text{lt} : X \to X \to \mathit{Bool}\ \}$
inst $\mathit{Ord}\ (\mathit{Int})\ \{\ \text{lt} = (<)\ \}$
```

The class definition declares a new class (dictionary) $\mathit{Ord}\ (X)$ that contains a method $\mathit{Ord}(X)$ that depends on a dictionary $\mathit{Eq}(X)$ and contains a method $\text{lt} : X \to X \to \mathit{Bool}$. The instance definition builds a dictionary $\mathit{Ord}(\mathit{Int})$ from the existing dictionary $\mathit{Eq}\ \mathit{Int}$ and the primitive $(<)$ for lt. The two declarations are elaborated into:

```
type $\mathit{Ord}\ (X) = \{\ \mathit{Eq} : \mathit{Eq}\ (X);\ \text{lt} : X \to X \to \mathit{Bool}\ \}$
let $\text{EqOrd}\ X\ (\text{OrdX} : \mathit{Ord}\ X) : \mathit{Eq}\ X = \text{OrdX}.\mathit{Eq}$
let $\text{lt}\ X\ (\text{OrdX} : \mathit{Ord}\ X) : X \to X \to \mathit{Bool} = \text{OrdX}.\text{lt}$
let $\text{OrdInt} : \mathit{Ord}\ \mathit{Int} = \{\ \mathit{Eq} = \text{EqInt};\ \text{lt} = (<)\ \}$
```

So far, we have just defined type classes and some instances. We may write a function that uses these overloaded definitions. When overloading cannot be resolved statically, the function will be abstracted other one or several additional arguments, called dictionnaries, that will carry the appropriate definitions for the unresolved overloaded symbols. For example, consider the following definition in Mini Haskell:

```
let rec $\mathit{search} : \forall(X)\ \mathit{Ord}\ X \Rightarrow X \to \mathit{List}\ X \to \mathit{Bool} =$
  $\Lambda(X)\ \lambda(x : X)\ \lambda(l : \mathit{List}\ X)$
    match $l$ with $[] \to \mathit{false} \mid h{::}t \to \text{equal}\ x\ h \mid\mid (\text{lt}\ \ h\ x\ \&\&\ \mathit{search}\ X\ x\ t)$
```

This code is elaborated into:

```
let rec $\mathit{search}\ X\ (\text{OrdX} : \mathit{Ord}\ X)\ (x : X)\ (l : \mathit{List}\ X) : \mathit{Bool} =$
  match $l$ with $[] \to \mathit{false}$
   $\mid\ h{::}t \to \text{equal}\ X\ (\text{EqOrd}\ X\ \text{OrdX})\ x\ h \mid\mid (\text{lt}\ X\ \text{OrdX}\ h\ x\ \&\&\ \mathit{search}\ X\ \text{OrdX}\ x\ t)$
```

Using the overloading function, as in $\mathit{search}\ \mathit{Int}\ 1\ [1;\ 2;\ 3]$ will then elaborate into the code $\mathit{search}\ \mathit{Int}\ \text{OrdInt}\ 1\ [1;\ 2;\ 3]$ where a dictionary OrdInt of the appropriate type has been built and passed as an additional argument. Here, the target language is the explicitly-typed System F, which has a type erasing semantics, hence the type argument Int may be dropped while the dictionary argument OrdInt is retained: the code that is actually executed is thus $\mathit{search}\ \text{OrdInt}\ 1\ [1;\ 2;\ 3]$ (where type information has been stripped off OrdInt itself).

7.2.2 The definition of Mini Haskell

Class declarations and instance definitions are restricted to the toplevel. Their scope is the whole program. In practice, a program p is a sequence of class declarations and instance and function definitions given in any order and ending with an expression. For simplification,

$$\begin{array}{rcl} p & ::= & H_1 \dots H_p \, h_1 \dots h_q \, M \\ H & ::= & \mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\} \\ \rho & ::= & u_1 : \tau_1, \dots u_n : \tau_n \\ h & ::= & \mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\} \\ r & ::= & u_1 = M_1, \dots u_n = M_n \end{array} \qquad \begin{array}{rcl} P & ::= & \mathsf{K}\ \alpha \\ \vec{P} & ::= & P_1, \dots P_n \\ Q & ::= & \mathsf{K}\ \tau \\ \vec{Q} & ::= & Q_1, \dots Q_n \\ \sigma & ::= & \forall \vec{\alpha}. \vec{Q} \Rightarrow T \\ T & ::= & \tau \mid Q \end{array}$$

Figure 7.1: Syntax of MH expressions and types

we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

Instance definitions are interpreted recursively and their definition order does not matter. We may assume, *w.l.o.g.*, that instance definitions come after all class declarations. The order of class declaration matters, since they may only refer to other class constructors that have been previously defined.

For sake of simplification, we restrict to single parameter classes. The syntax of MH programs is defined in Figure 7.1. Letter p ranges over source programs. A program p is a sequence $H_1 \dots H_p\ h_1 \dots h_q\ M$, of class declaration $H_1 \dots H_p$, followed by a sequence of instance definitions $h_1 \dots h_q$, and ending with an expression M.

A class declaration H is of the form $\mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$. It defines a new class (constructor) K, parametrized by α. Every class (constructor) K must be defined by one and only one class declaration. So we may say that H is the declaration of K and write H_{K}.

Letter u ranges over *overloaded symbols*, also called *methods*. The row ρ of the form $u_1 : \tau_1, \dots u_n : \tau_n$ declares overloaded symbols u_i of class K. An overloaded symbol cannot be declared twice in a program; it cannot be repeated twice in the same class (hence the map $i \mapsto u_i$ is injective) and cannot be declared in two different classes. The row ρ (and thus each of its field type τ_i) must not contain any other free variable than α.

The class depends on a sequence of subclasses $\vec{P}$ of the form $\mathsf{K}_1\ \alpha, \dots \mathsf{K}_n\ \alpha$, which is called a *typing context*. Each clause $\mathsf{K}_i\ \alpha$ can be read as an assumption *"given an instance of class* K_1 *at type* α*"* and $\vec{P}$ as the conjunction of these assumptions. We say that classes K_i's are superclasses of K which we write $\mathsf{K}_i \prec \mathsf{K}$. They must have been previously defined. This ensures that the relation $\prec$ is acyclic. We require that all K_i's are independent, *i.e.* there does not exists i and j such that $\mathsf{K}_j \prec \mathsf{K}_i$.

An instance definition h is of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$. It defines an instance of a class K at type $\mathsf{G}\ \vec{\beta}$ where G is a datatype constructor, *i.e.* neither an arrow type nor a class constructor. A class constructor K may appear in T but not in τ. An instance definition *defines* the methods of a class at the required type: r is a record of methods

$u_1 = M_1, \ldots u_n = M_n$.

An instance definition is also parametrized by a typing context $\vec{P}$ of the form $K_1\ \alpha_1, \ldots K_k\ \alpha_k$ where variables α_i's are included in $\vec{\beta}$. This typing context is not related to the typing context of its class declaration H_{K}, but to the set of classes that the implementations of the methods depend on.

Restrictions The restriction to types of the form $\mathsf{K}'\ \alpha'$ in typing contexts and class declarations, and to types of the form $\mathsf{K}'\ (\mathsf{G}'\ \vec{\alpha}')$ in instances are for simplicity. Generalization are possible and discussed later (§7.4).

7.2.3 Semantics of Mini Haskell

The semantics of Mini Haskell is given by elaborating source programs into System F extended with record types and recursive definitions. Record types are provided as data types. They are used to represent dictionaries. Record labels are used to encode overloaded identifiers u. We may use overloaded symbols as variables as well: this amounts to reserving a subset of variables x_u indexed by overloaded symbols and writing u as a shortcut for x_u. We use letter N instead of M for elaborated terms, to distinguish them from source terms. For convenience, we write $\Rightarrow$ in System F as an alias for $\rightarrow$, which we use when the argument is a (record representing a) dictionary. Type schemes in the target language take the form σ described on Figure 7.1. Notice that types T are stratified: they are either dictionary types $\mathsf{K}\ \tau$ or a regular type τ that does not contain dictionary types.

Class declaration The elaboration of a class declaration H_{K} of the form $\mathsf{class}\ \mathsf{K}_1\ \alpha, \ldots \mathsf{K}_n\ \alpha \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$ consists of several parts. It first declares a record type that will be used as a dictionary to carry both the methods and the dictionaries of its *immediate* superclasses. A class need not contain subdictionaries recursively, since if $\mathsf{K}_j \prec \mathsf{K}_i$, then a dictionary for K_i already contains a sub-dictionary for K_j, to which K has access via K_i so it does need not have one itself. The row ρ of the class definition only lists the class methods. Hence, we extend it with fields for sub-dictionaries and define the record type:

$$\mathsf{K}\ \alpha \approx \{\rho^{\mathsf{K}}\} \qquad \text{where } \rho^{\mathsf{K}} \text{ is } u^{\mathsf{K}}_{\mathsf{K}_1} : \mathsf{K}_1\ \alpha, \ldots u^{\mathsf{K}}_{\mathsf{K}_n} : \mathsf{K}_n\ \alpha, \rho.$$

This record type declaration is collected to appear in the program *prelude.*

Then, for each $u : T_u$ in ρ^{K}, we define the program context:

$$\mathcal{R}_u \triangleq \mathsf{let}\ u : \sigma_u = N_u\ \mathsf{in}\ [\,] \qquad \text{where} \quad \sigma_u \triangleq \forall \alpha.\, \mathsf{K}\ \alpha \Rightarrow T_u \ \text{ and } \ N_u \triangleq \Lambda\alpha.\lambda z{:}\mathsf{K}\ \alpha.\, (z.u)$$

Let the composition $\mathcal{R}_1 \circ \mathcal{R}_2$ of two contexts be the context $\mathcal{R}_1[\mathcal{R}_2]$ obtained by placing $\mathcal{R}_2$ in the hole of $\mathcal{R}_1$. The elaboration $[\![H_{\mathsf{K}}]\!]$ of a single class declaration H_{K} is the composition:

$$[\![H_{\mathsf{K}}]\!] \triangleq \mathcal{R}_{u_1} \circ \ldots \mathcal{R}_{u_n} \qquad \text{where} \quad \mathsf{K}\ \alpha \approx \{u_1 : T_1, \ldots u_n : T_n\}$$

that defines accessors for each field of the class dictionary. We also define the typing environment $\Gamma_{H_{\mathsf{K}}}$ as an abbreviation for $u_1 : \sigma_{u_1}, \dots u_n : \sigma_{u_n}$.

The elaboration $[\![H_1 \dots H_p]\!]$ of all class definitions is the composition $[\![H_1]\!] \circ \dots [\![H_p]\!]$ of the elaboration of each. We also define $\Gamma_{H_1 \dots H_n}$ as the concatenation $\Gamma_{H_1}, \dots \Gamma_{H_n}$ of individual typing environments.

Instance definition In an instance declaration h of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$, The typing context $\vec{P}$ describes the dictionaries that must be available on type parameters $\vec{\beta}$ for constructing the dictionary $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, but that cannot yet be built because they depend on some unknown type β in $\vec{\beta}$.

As mentioned above $\vec{P}$ is not related to the typing context of the class declaration H_{K}. To see this, assume that class K' is an immediate superclass of K, so that the creation of the dictionary $\mathsf{K}\ \alpha$ requires the existence of a dictionary $\mathsf{K}'\ \alpha$; then, an instance declaration $\mathsf{K}\ \mathsf{G}$ (where G is nullary) need not be parametrized over a dictionary of type $\mathsf{K}'\ \mathsf{G}$, as either such a dictionary can already be built, hence the instance definition does not require it, or it will never be possible to build one, as instance definitions are recursively defined so all of them are already visible—and the program must be rejected.

We restrict typing context $\mathsf{K}_1\ \alpha_1, \dots \mathsf{K}_k\ \alpha_k$ to *canonical* ones defined as satisfying the two following conditions: (1) α_i is some β_j in $\vec{\beta}$; and (2) if K_i and K_j are related, *i.e.* $\mathsf{K}_i \prec \mathsf{K}_j$ or $\mathsf{K}_j \prec \mathsf{K}_i$ or $\mathsf{K}_i = \mathsf{K}_j$. then α_i and α_j are different. The latter condition avoids having two dictionaries $\mathsf{K}_i\ \beta$ and $\mathsf{K}_j\ \beta$ when, *e.g.*, $\mathsf{K}_i \prec \mathsf{K}_j$ since the former is contained in the latter.

The elaboration of an instance declaration h is a triple (z_h, N^h, σ_h) where z_h is an identifier to refer to the elaborated body N^h of type

$$\sigma_h \ \triangleq\ \forall \beta_1 \dots \beta_p . \mathsf{K}_1\ \alpha_1 \Rightarrow \dots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$$

(Variables $\alpha_1, \dots \alpha_k$ are among $\beta_1, \dots \beta_p$ and may contain repetitions, as explained above.) The expression N^h builds a dictionary of type $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, given k dictionaries (where k may be *zero*) of respective types $\mathsf{K}_1\ \beta_1, \dots \mathsf{K}_k\ \beta_k$ and is defined as:

$$\begin{aligned} N^h \ \triangleq\ & \Lambda\beta_1. \dots \Lambda\beta_p.\ \lambda(z_1 : \mathsf{K}_1\ \alpha_1). \dots \lambda(z_k : \mathsf{K}_k\ \alpha_k). \\ & \qquad \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \dots u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N^h_1, \dots u_m = N^h_m\} \end{aligned}$$

The types of fields are as prescribed by the class definition K, but specialized at type $\mathsf{G}\ \vec{\beta}$. That is, q_i is a dictionary expression of type $\mathsf{K}'_i\ (\mathsf{G}\ \vec{\beta})$ whose exact definition is postponed until the elaboration of dictionaries in §7.2.6. The term N^h_i is the elaboration of M_i where $u_1 = M_1, \dots u_m = M_m$ is r; it is described in the next section (§7.2.4). For clarity, we write z instead of x when a variable binds a dictionary or a function building a dictionary. Notice that the expressions q_i and N^h_i sees the type variables $\beta_1, \dots \beta_p$ and the dictionary parameters $z_1 : \mathsf{K}_1\ \alpha_1, \dots z_k : \mathsf{K}_k\ \alpha_k$.

The elaboration of all instance definitions is the program context:

$$[\![\vec{h}]\!] \triangleq \mathsf{let\ rec}\ (\vec{z}_h : \vec{\sigma}_h) = \vec{N}^h\ \mathsf{in}\ []$$

that recursively binds all instance definitions in the hole.

Program Finally, the elaboration of a complete program $\vec{H}\ \vec{h}\ M$ is

$$[\![\vec{H}\ \vec{h}\ M]\!] \triangleq ([\![\vec{H}]\!] \circ [\![\vec{h}]\!])[M] = \mathsf{let}\ \vec{u} : \vec{\sigma}_u = \vec{N}_u\ \mathsf{in\ let\ rec}\ (\vec{z}_h : \vec{\sigma}_h) = \vec{N}^h\ \mathsf{in}\ N$$

Hence, the expression N, which is the elaboration of M, and all expressions N_h are typed (and elaborated) in the environment $\Gamma_{\vec{H}\vec{h}}$ equal to $\Gamma_{\vec{H}}$, $\Gamma_{\vec{h}}$: the environment $\Gamma_{\vec{H}}$ declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols) while the environment $\Gamma_{\vec{h}}$, declares functions to build dictionaries.

7.2.4 Elaboration of expressions

The elaboration of expressions is defined by a judgment $\Gamma \vdash M \rightsquigarrow N : \sigma$ where Γ is a System F typing context, M is the source expression, N is the elaborated expression and σ its type in Γ. In particular, $\Gamma \vdash M \rightsquigarrow N : \sigma$ implies $\Gamma \vdash N : \sigma$ in System F.

We write q for dictionary terms, which are the following subset of System-F terms:

$$q ::= u \mid z \mid q\ \tau \mid q\ q$$

Variables u and z are just particular cases of variables x. Variable u is used for methods (and access to subdictionaries), while variable z is used for dictionary parameters and for class instances, *i.e.* dictionaries or functions building dictionaries.

The rules for elaboration of expressions are described in Figure 7.2. Most of them just wrap the elaboration of their sub-expressions. In rule Let, we require σ to be canonical, *i.e.* of the form $\forall \vec{\alpha}. \vec{P} \Rightarrow T$ where $\vec{P}$ is itself empty or canonical (see page 153). Rules App and Abs do not apply to overloaded expressions of type σ but only to simple expressions of type τ.

The interesting rules are the elaboration of overloaded expressions, and in particular of missing abstractions (Rule OAbs) and applications (Rule OApp) of dictionaries. Rule OAbs pushes dictionary abstractions in the context Γ as prescribed by the expected type. On the opposite, Rule OApp searches for an appropriate dictionary-building function and applies it to the required sub-directionary.

The premise $\Gamma \vdash q : Q$ of rule OApp also triggers the elaboration of dictionaries. This judgment is just the typability in System F—but restricted to dictionary expressions. That is, it searches for a well-typed dictionary expression. The restriction to dictionary expressions ensures that under reasonable conditions the search is decidable—and coherent. The elaboration of dictionaries reads the typing rules of System F restricted to dictionaries as an algorithm, where Γ and Q are given and q is inferred. This is described in detail in §7.2.6.

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \sigma} \qquad \textsc{Inst}\ \frac{\Gamma \vdash M \rightsquigarrow N : \forall \alpha.\, \sigma}{\Gamma \vdash M\ \tau \rightsquigarrow N\ \tau : [\alpha \mapsto \tau]\sigma} \qquad \textsc{Gen}\ \frac{\Gamma, \alpha \vdash M \rightsquigarrow N : \sigma}{\Gamma \vdash \Lambda\alpha.M \rightsquigarrow \Lambda\alpha.N : \forall \alpha.\, \sigma}$$

$$\textsc{Let}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 \rightsquigarrow N_2 : \tau}{\Gamma \vdash \mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2 \rightsquigarrow \mathsf{let}\ x : \sigma = N_1\ \mathsf{in}\ N_2 : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau' \vdash M \rightsquigarrow N : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, M \rightsquigarrow \lambda x{:}\tau'.\, N : \tau' \to \tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 \rightsquigarrow N_2 : \tau_2}{\Gamma \vdash M_1\ M_2 \rightsquigarrow N_1\ N_2 : \tau_1} \qquad \textsc{OAbs}\ \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda x{:}Q.\, N : Q \Rightarrow \sigma}$$

$$\textsc{OApp}\ \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

Figure 7.2: Elaboration of expressions

By construction, elaboration produces well-typed expressions: that is $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \tau$ implies that is $\Gamma_{\vec{H}\vec{h}} \vdash N : \tau$.

7.2.5 Summary of the elaboration

An instance declaration h of the form:

$$\mathsf{inst}\ \forall \vec{\beta}.\ \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ \vec{\tau}\ \{u_1 = M_1, \ldots l; u_m = M_m\}$$

is translated into

$$\lambda(z_1{:}\mathsf{K}_1\ \alpha_1) \ldots .\, \lambda(z_p{:}\mathsf{K}_k\ \alpha_k).\, \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots\ u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N_1, \ldots\ u_m = N_m\}$$

where $u^{\mathsf{K}}_{\mathsf{K}'_i} : \tau_i$ are the superclasses fields, Γ^h is $\vec{\beta}, \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k$, and the following elaboration judgments $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash q_i : \tau_i$ and $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash M_i \rightsquigarrow N_i : \tau_i$ hold. Finally, given the program p equal to $\vec{H}\ \vec{h}\ M$, we elaborate M as N such that $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \forall \bar{\alpha}.\, \tau$.

Notice that $\forall \bar{\alpha}.\, \tau$ is an unconstrained type scheme. Otherwise, N could elaborate into an abstraction over dictionaries, which could turn a computation into a function that is not reduced: this would not preserve the intended semantics.

More generally, we must be careful to preserve the *intended* semantics of source programs. For this reason, in a call-by-value setting, we must not elaborate applications into abstractions, since this could delay and perhaps duplicate the order of evaluations. We just pick the obvious solution, that is to restrict rule Let so that either σ is of the form $\forall \bar{\alpha}.\, \tau$ or M_1 is a value or a variable.

In a language with a call-by-name semantics, an application is not evaluated until it is needed. Hence adding an abstraction in front of an application should not change the evaluation order $M_1\ M_2$. We must in fact compare:

$$\mathsf{let}\ x_1 = \lambda y.\,\mathsf{let}\ x_2 = V_1\ V_2\ \mathsf{in}\ M_2\ \mathsf{in}\ [x_1 \mapsto x_1\ q]M_1 \qquad (1)$$
$$\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \lambda y.\,V_1\ V_2\ \mathsf{in}\ [x_2 \mapsto x_2\ q]M_2\ \mathsf{in}\ M_1 \qquad (2)$$

The order of evaluation of $V_1\ V_2$ is preserved. However, the Haskell language is call-by-need and not call-by-name! Hence, applications are delayed as in call-by-name but shared and only reduced once. The application $V_1\ V_2$ will be reduced once in (1), but as many times as there are occurrences of x_2 in M_2 in (2).

The final result will still be the same in both cases if the language has no side effects, but the intended semantics may be changed regarding the complexity.

Coherence The elaboration may fail for several reasons: The input expression may not obey one of the restrictions we have requested; a typing error may occur during elaboration of an expression; or or some dictionary cannot be build. If elaboration fails, the program p is rejected, of course.

When the elaboration of p succeeds, it should return a term $[\![p]\!]$ that is well-typed in F and that defines the semantics of p. However, although terms are explicitly-typed, their elaboration may not be unique! Indeed, they might be several ways to build dictionaries of some given type, as we shall see below (§7.2.6).

We may distinguish two situations: in the worst case, a source program may elaborate to several completely unrelated programs; in the better case, all possible elaborations may in fact be *equivalent* programs: we say that the elaboration is *coherent* and the programs has a deterministic semantics given by any of its elaboration.

Opening a parenthesis, what does it mean for programs be equivalent? There are several notions of program equivalence:

- If programs have a denotational semantics, the equivalence of programs should be the equality of their denotations.
- As a subcase, two programs having a common reduct should definitely be equivalent. However, this will in general not be complete: values may contain functions that are not identical, but perhaps reduce to the same value whenever applied to equivalent arguments.
- This leads to the notion of *observational equivalence.* Two expressions are observationally equivalent (at some observable type, such as integers) if their are indistinguishable whenever they are put in arbitrary (well-typed) contexts of the observable type.

End of parenthesis.

$$\textsc{D-OVar}\quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{D-Inst}\quad \frac{\Gamma \vdash q : \forall \alpha.\, \sigma}{\Gamma \vdash q\, \tau : [\alpha \mapsto \tau]\sigma} \qquad \textsc{D-App}\quad \frac{\Gamma \vdash q_1 : Q_1 \Rightarrow Q_2 \qquad \Gamma \vdash q_2 : Q_1}{\Gamma \vdash q_1\, q_2 : Q_2}$$

Figure 7.3: Typing rules for dictionaries

For instance, two different elaborations algorithms that consistently change the representation of dictionaries (*e.g.* by ordering records in reverse order), may be equivalent if we cannot observe the representation of dictionaries.

Returning to the coherence problem, the only source of non-determinism in Mini Haskell is the elaboration of dictionaries. Hence, to ensure coherence, it suffices that two dictionary *values* of the same type are always equal. This does not mean that there is a unique way of building dictionaries, but that all ways are equivalent as they eventually return the same dictionary.

7.2.6 Elaboration of dictionaries

The elaboration of dictionaries is based on typing rules of System F—but restricted to a subset of the language. The relevant typing rules are given in Figure 7.3. However, elaboration significantly differs from type inference since the judgment $\Gamma \vdash q : Q$ is used for inferring q rather than τ. The judgment can be read as: in type environment Γ, a dictionary of type Q can be constructed by the dictionary expression q. As for type inference, elaboration of dictionaries is simplified by finding an appropriate syntax-directed presentation of the typing rules—but directed by the structure of the type of the expected dictionary instead of expressions.

Elaboration is also driven by the bindings available in the typing environment. These may be dictionary constructors z^h, given by instance definitions; dictionary accessors u^{K}, given by class declarations; dictionary arguments z, given by the local typing context. This suggests the presentation of the typing rules in Figure 7.4.

Dictionary values Let us first consider the elaboration of dictionary *values*. They are typed in the environment $\Gamma_{\vec{H}\vec{h}}$, which does not contain free *type* variables. Hence, rule D-Var does not apply. Moreover, dictionaries stored in other dictionaries had to be built in the first place, hence rule D-Proj should never be needed. That is, dictionary values can be built with only instances of D-OVar-Inst of the form:

$$\textsc{D-OVar-Inst}\quad \frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\, P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma_{\vec{h}} \qquad \Gamma_{\vec{h}} \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma_{\vec{h}} \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}$$

$$\textsc{D-OVar-Inst}\quad \frac{z : \forall\vec{\beta}.\, P_1 \Rightarrow \dots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma \qquad \forall i \in 1..n,\ \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}$$

$$\textsc{D-Proj}\quad \frac{u : \forall\alpha.\, \mathsf{K}'\ \alpha \Rightarrow \mathsf{K}\ \alpha \in \Gamma \qquad \Gamma \vdash q : \mathsf{K}'\ \tau}{\Gamma \vdash u\ \tau\ q : \mathsf{K}\ \tau} \qquad \textsc{D-Var}\quad \frac{z : \mathsf{K}\ \alpha \in \Gamma}{\Gamma \vdash z : \mathsf{K}\ \alpha}$$

Figure 7.4: Algorithmic typing rules for dictionaries

where the premises $\Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i$ are themselves recursively built in the same way. This rule can be read as a recursive definition, where Γ is constant, Q is the input type of the dictionary, and q is the output dictionary. This reading is deterministic if there is no choice in finding $z : \forall\vec{\beta}.\, P_1 \Rightarrow \dots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$ in Γ. The binding z can only be a binding z^h introduced as the elaboration of some class instance h at type $\Gamma\vec{\beta}$. Hence, it suffices that *instance definitions never overlap* for z^h to be uniquely determined; if recursively each q_i is unique, then $z\ \vec{\tau}\ \vec{q}$ also is. Under this hypothesis, the elaboration is always unique and therefore coherent.

Definition 3 (Overlapping instances) *Two instances* inst $\forall\vec{\beta}_1.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_1\ \vec{\beta}_1)\ \{r_1\}$ *and* inst $\forall\vec{\beta}_2.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_2\ \vec{\beta}_2)\ \{r_2\}$ *of a class* K overlap *if the type schemes* $\forall\vec{\beta}_1.\mathsf{K}\ (\mathsf{G}_1\ \vec{\tau}_1)$ *and* $\forall\vec{\beta}_2.\mathsf{K}\ (\mathsf{G}_2\ \vec{\tau}_2)$ *have a common instance,* i.e. *in the current setting, if* G_1 *and* G_2 *are equal.*

Overlapping instances are an inherent source of incoherence, as it means that for some type Q (in the common instance), a dictionary of type Q may (possibly) be built using two different implementations.

Dictionary expressions Dictionary expressions may compute on dictionaries: they may extract sub-dictionaries or build new dictionaries from other dictionaries received as argument. Indeed, in overloaded code, the exact type is not fully known at compile type, hence dictionaries must be passed as arguments, from which superclass dictionaries may be extracted (actually must be extracted, as we forbade to pass a class and one of its super class dictionaries simultaneously).

Dictionaries are typically typed in the typing environment $\Gamma_{\vec{H}\vec{h}}, \Gamma^h$ where Γ^h binds the local typing context, *i.e.* assumptions $z : \mathsf{K}'\ \beta$ about dictionaries received as arguments. Hence, rules D-Proj and D-Var may now apply, *i.e.* the elaboration of expressions uses the three rules of 7.4. This can still be read as a backtracking proof search algorithm. The proof search always terminates, since premises always have strictly smaller Q than the conclusion when using the lexicographic ordering of the height of τ and then the reverse order of class inheritance: when no rule applies, the search fails; when rule D-Var applies, the search ends with a successful derivation; when rule D-Proj applies, the premise is called with a smaller problem since the height is unchanged and $\mathsf{K}' \, \vec{\tau}$ with $\mathsf{K}' \prec \mathsf{K}$; when D-Ovar-Inst applies, the premises are called at type $\mathsf{K}_i \, \tau_j$ where τ_j is subtype of $\vec{\tau}$, hence of a strictly smaller height.

Non determinism However, non-overlapping of class instances is no more sufficient to prevent non determinism. For instance, the introductory example of §7.2.1 defines two instances EqInt and OrdInt where the later contains an instance of the former. Hence, a dictionary of type EqInt may be obtained, either directly as EqInt, or indirectly as *Eq* OrdInt, by projecting the *Eq* sub-dictionary of class *Ord Int*. In fact, the latter choice could then be reduced at compile time and be equivalent to the first one.

One could force more determinism by fixing a strategy for elaboration. Restrict the use of rule D-Proj to cases where Q is P–when D-OVar-Inst does not apply. However, since the two elaborations paths are equivalent, the extra flexibility is harmless and may perhaps be useful freedom for the compiler.

Example of elaboration In our introductory example, the typing environment $\Gamma_{\vec{H}\vec{h}}$ is (we remind both the informal and formal names of variables):

$$\begin{array}{rcll}
\text{equal} & \triangleq & u_{equal} & : \forall \alpha.\, Eq\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}, \\
\text{EqInt} & \triangleq & z^{Int}_{Eq} & : Eq\ \mathsf{int} \\
\text{EqList} & \triangleq & z^{List}_{Eq} & : \forall \alpha.\, Eq\ \alpha \Rightarrow Eq\ (\mathsf{list}\ \alpha) \\
\\
\text{EqOrd} & \triangleq & u^{Ord}_{Eq} & : \forall \alpha.\, Ord\ \alpha \Rightarrow Eq\ \alpha \\
\text{lt} & \triangleq & u_{lt} & : \forall \alpha.\, Ord\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}
\end{array}$$

When elaborating the body of the *search* function, we have to infer a dictionary for EqOrd X OrdX in the local context X, OrdX : *Ord* X. Using formal notations, dictionaries are typed in the environment Γ equal to $\Gamma_0, \alpha, z : Ord\ \alpha$. and EqOrd is u^{Ord}_{Eq}. We have the following derivation:

$$\textsc{D-Proj}\ \frac{\overset{\textsc{D-OVar-Inst}}{\Gamma \vdash z : u^{Ord}_{Eq} : Ord\ \alpha \to Eq\ \alpha} \qquad \overset{\textsc{D-Var}}{\Gamma \vdash z : Ord\ \alpha}}{\Gamma \vdash u^{Ord}_{Eq}\ \alpha\ z : Eq\ \alpha}$$

7.3 Implicitly-typed terms

Our presentation of Mini Haskell is explicitly typed. Since we remain within an ML-like type system where type schemes are not first-class, we may leave some type information implicit. But how much? Class declarations define both the structure of dictionaries—a record type definition and its accessors—and the type scheme of overloaded symbols. Since, we inferring type schemes is out of the scope of ML-like type inference, class declarations

5 Solutions

Question 1

```
let return v = fun () → Some v
let fail = fun () → None
let run s = s ()
```

Question 2

```
let map f s = fun () →
  match s () with
  | None → None
  | Some v → Some (f v)

let pair s1 s2 = fun () → match s1 (), s2 () with
| Some v1, Some v2 → Some (v1, v2)
| None, _ | _, None → None

let bind s f = fun () →
  match s () with
  | None → None
  | Some v → run (f v)
```

Note. *Many student answers used definition that are "too strict", in the following style:*

```
let map f s =
  match run s with
  | None → fail
  | Some v → return (f v)

let pair s1 s2 = match run s1, run s2 with
| Some v1, Some v2 → return (v1, v2)
| None, _ | _, None → fail

let bind s f =
  match run s with
  | None → fail
  | Some v → f v
```

These answers are not completely wrong but they are still incorrect, in that they force the computation of the search "right away". For example, the result of `map f s` *should be a search procedure that,* once run, *will run* `s` *and apply* `f` *to the result, but the strict definition above runs* `s` *right away.*

In particular, with these definitions, the computational cost of

```
  sum (map f1 s1) (map f2 s2)
```

is at least `cost(s1) + cost(s2)` *in a strict language, no matter how* `sum` *is defined. This breaks the expectation that at most one of* `s1` *and* `s2` *will be run if they both succeed. This degrades the algorithmic complexity of many search programs.*

Question 3

```
  let sum s1 s2 = fun () →
    match run s1 with
    | Some v1 → Some v1
    | None → run s2
```

```
let rec traverse = function
| [] → return []
| x :: xs →
  let+ v = x
  and+ vs = traverse xs
  in v :: vs
```

Question 4

```
  run (let+ x = fail in t)
= match run fail with
  | None → None
  | Some x → (fun x → t) x
= match None with
  | None → None
  | Some x → t
= None
= run fail

run (fail ++ t)
= match run fail with
  | Some v → Some v
  | None → run t
= match None with
  | Some v → Some v
  | None → run t
= run t

run (t ++ fail)
= match run t with
  | Some v → Some v
  | None → run fail
= match run t with
  | Some v → Some v
  | None → None
= run t
```

Question 5

```
let int = (Constr "int", [])
```

Question 6

```
let ord_alpha = (Class "Ord", TyVar "alpha")
let eq_int = (Class "Eq", int)
```

Question 7

```
Proj (TeVar "u", int, Instance (TeVar "i", [], []))
```

Note. *Several student answers missed the* `Instance` *constructor and instead used directly*

```
Proj (TeVar "u", int, TeVar "i")}
```

This is not correct, as a raw `TeVar "i"` *corresponds to the judgment* D-VAR *of the algorithmic typing rules, which is reserved for local class hypotheses at types of the form* $K\ \alpha$.

Question 8

```
let inst_ord_int = {
  params = [];
  depndencies = [];
  class_name = Class "Ord";
  constr_name = Constr "Int";
}
```

Question 9

```
{
  projections = [(TeVar "u", Class "Ord", Class "Eq")];
  instances = [(TeVar "i", inst_ord_int)];
  tyvars = [TyVar "alpha"];
  vars = [(TeVar "z", ord_alpha)];
}
```

Question 10

```
let guard : bool → unit search = function
| true → return ()
| false → fail

let rec find_map (f : (α → β search)) (li : α list) : β search =
  fun () → List.find_map (fun v → run (f v)) li

let apply_subst subst (k, alpha) =
  let tau =
    try List.assoc alpha subst
    with Not_found → TyVar alpha
  in (k, tau)

(* $\Gamma \vdash q : Q$ *)
let rec elaborate (env : env) (goal : typ class_expr) : term search =
  begin
    (* $\Gamma \vdash z\ \bar{\tau}\ \bar{q} : K\ (G\ t\bar{a}u)$ *)
    match goal with
    | (k, TyVar _) → fail
    | (k, TyConstr (g, taus)) →
    env.instances |> find_map @@ fun (z, inst) →
    let* () = guard (inst.class_name = k && inst.constr_name = g) in
    (* $z : \forall\bar{\beta}.\ \bar{P} \Rightarrow K\ (G\ \bar{\beta})$ *)
    let subst = (* $[\bar{\tau}/\bar{\beta}]$ *)
      List.map2 (fun beta tau → (beta, tau)) inst.params taus in
    let+ qs = (* $\Gamma \vdash q : [\bar{\tau}/\bar{\beta}]\bar{P}$ *)
      inst.dependencies
      |> List.map (fun p → elaborate env (apply_subst subst p))
      |> traverse in
    Inst (z, taus, qs)
  end
  ++
  begin
    (* $\Gamma \vdash u\ \tau\ q : K\ \tau$ *)
    let (k, tau) = goal in
    env.projections |> find_map @@ fun (u, k_from, k_to) →
    let* () = guard (k_to = k) in
```

```
          let+ q = elaborate env (k_from, tau) in
          Proj (u, tau, q)
        end
        ++
        begin
          (* Γ ⊢ z : K  α  *)
          match goal with
          | (k, TyConstr _) → fail
          | (k, TyVar alpha) →
          env.vars |> find_map @@ fun (z, (k', alpha')) →
          let+ () = guard (k = k' && alpha = alpha') in
          Var z
        end
```

Question 11

1. `eq_prod eq_int eq_string`
2. `eq_of_ord (ord_prod ord_int ord_string)`

Question 12

```
let eq_int = { equal = equal_int }
let eq_prod ea eb = {
  equal = equal_prod ea.equal eb.equal;
}

let ord_int = { eq = eq_int; less = less_int }
let ord_prod oa ob = {
  equal = eq_prod oa.eq ob.eq;
  less = less_prod oa.less ob.less;
}
```

Question 13

```
eq_of_ord (ord_prod ord_int ord_string)
= (fun inst → inst.eq) {
    eq = eq_prod ord_int.eq ord_string.eq;
    less = less_prod ord_int.less ord_string.less;
  }
= eq_prod ord_int.eq ord_string.eq
= eq_prod eq_int eq_string
```

Question 14

```
val set_bool : (bool, bitset) set
```

Question 15

```
val set_of_list : (α, 'repr) set → α list → 'repr
```

Question 16

```
val int_map : (int, patricia_trie) map
val map_of_list : (α, 'repr) map → (α * β) list → β repr
```