Modules – Low road

Better error messages via diffing

Florian Angeletti and Gabriel Radanne

Error messages for modules

Problem: Modules errors are often verbose and complicated

Remark: Modules are often shallow and their structure is "list-like"!

Idea: Use the semantic structure of modules for "diffing"

Error messages for modules

Problem: Modules errors are often verbose and complicated

Remark: Modules are often shallow and their structure is "list-like"!

Idea: Use the semantic structure of modules for "diffing"

Error messages for modules

Problem: Modules errors are often verbose and complicated

Remark: Modules are often shallow and their structure is “list-like”!

Idea: Use the semantic structure of modules for “diffing”

```
module Graph(Vertex:VERTEX)(Edge:EDGE) = struct ... end

module G = Graph(Label)(Vertex)(Edge)
```

```
module Graph(Vertex:VERTEX)(Edge:EDGE) = struct ... end

module G = Graph(Label)(Vertex)(Edge)

Error: Signature mismatch:
       Modules do not match:
         sig type t = string end
       is not included in
         VERTEX
       The value `label' is required but not provided
       The value `create' is required but not provided
       The type `label' is required but not provided
       The value `equal' is required but not provided
       The value `hash' is required but not provided
       The value `compare' is required but not provided
```

```
module Graph(Vertex:VERTEX)(Edge:EDGE) = struct ... end

module G = Graph(Label)(Vertex)(Edge)

Error: The functor application is ill-typed.
       These arguments:
         Label Vertex Edge
       do not match these parameters:
         functor  (Vertex : VERTEX) (Edge : EDGE) -> ...
  1. The following extra argument is provided
      Label : sig type t = string end
  2. Module Vertex matches the expected module type VERTEX
  3. Module Edge matches the expected module type EDGE
```

Finished work:

Full implementation for **functor applications** `F(A)(B)(C)`

- Detects additions/deletions/mismatch (Levenstein distance)
- Solid semantic footing (errors are *correct* and *useful*)
- Scales very well in practice
- Handles the full complexity of OCaml modules (dependent, variadic, ...)

Implementation in **OCaml PR9331**. Review in progress.

Presented at **ML workshop 2020**
"High-level error messages for modules through diffing"

Ongoing/future work:

Apply diffing to other structural "diffable" types:

- **Signatures!**
  ⇒ Boilerplate done, "just" need to write the error messages
- objects/polymorphic variants
  ⇒ Easy in theory
- Function applications
  ⇒ promising, but much harder due to unification/inference
- Tree diffing for complete module hierarchies
  ⇒ Probably overkill

Modules – High road

Incremental modules

Gabriel Radanne, Thomas Refis, Jacques Garrigue and Didier Remy

Status of the module system – The good

Core system has excellent theoretical footing

- Applicative functor – `Map.Make(String)`
- Module ascription – `(M : S)`
- Manifest – `type foo = Foo.t = MyFoo`
- Qualified access – `A.v`, `F(X).t`

✔ All well understood/formalized

Status of the module system – The less good

What about “recent” features ?

- Generative functor: Well understood ✔
- `with` constraints: No real formalization ✘
- `module type of`: No formalization, Uncertain semantics ✘
- First class modules: Formalized as “package types” ✔... but only in SML ✘
- Module aliases: Well studied in theory as singletons ✔... but in very different settings ✘
- Private fields: Only partial formalization
- Interactions are not always clear

Status of the module system – The less good

What about “recent” features ?

- Generative functor: Well understood ✔
- `with` constraints: No real formalization ✘
- `module type of`: No formalization, Uncertain semantics ✘
- First class modules: Formalized as “package types” ✔... but only in SML ✘
- Module aliases: Well studied in theory as singletons ✔... but in very different settings ✘
- Private fields: Only partial formalization
- Interactions are not always clear

Another look at the module system

Goal: Let's have a new look at module formalization:

- Reconcile the problematic pieces
- Think about inference
- Fix/improve the known problems
- Provide a sure footing for further experimentation (modular implicits, . . .)

Non-goal: Design a new language. 1ML is cool, but we can't use it.

Let's look at some of the big problems!

Another look at the module system

Goal: Let's have a new look at module formalization:

- Reconcile the problematic pieces
- Think about inference
- Fix/improve the known problems
- Provide a sure footing for further experimentation (modular implicits, ...)

Non-goal: Design a new language. 1ML is cool, but we can't use it.

Let's look at some of the big problems!

Another look at the module system

Goal: Let's have a new look at module formalization:

- Reconcile the problematic pieces
- Think about inference
- Fix/improve the known problems
- Provide a sure footing for further experimentation (modular implicits, ...)

Non-goal: Design a new language. 1ML is cool, but we can't use it.

Let's look at some of the big problems!

Problem: It is not sound to create an alias of a functor parameter:

```
module F (X : S) = struct
  module A = X (* this is not an alias *)
end
```

Why: if the alias is kept, that might lead to segfaults?!?.

Problem: It is not sound to create an alias of a functor parameter:

```
module F (X : S) = struct
  module A = X (* this is not an alias *)
end
```

Why: if the alias is kept, we could discover that `F(MyModule).A = MyModule`, and thus discover fields that are not in `S`. Since module coercions are implemented by copies, that might lead to segfaults.

Problem: It is not sound to create an alias of a functor parameter:

```
module F (X : S) = struct
  module A = X (* this is not an alias *)
end
```

Why: if the alias is kept, we could discover that `F(MyModule).A = MyModule`, and thus discover fields that are not in `S`. Since module coercions are implemented by copies, that might lead to segfaults.

Currently: The type is expanded and module equality is lost:

```
module F (X : S) : sig
  module A : S with type t = X.t
end
```

Transparent ascriptions

Solution: Transparent ascriptions!

Transparent ascriptions hide the values but keep the types.

```
module F (X : S) : sig
  module A = (X <: S) (* X viewed through S *)
end
```

- A "filter" on the dynamic part of the module
- Also in path : `F(Y <: S).A`
- Generally useful for "soft constraints":
  `module MyKey <: Map.OrderedType = ...`
- Essential for modular implicits

Transparent ascriptions

Solution: Transparent ascriptions!

Transparent ascriptions hide the values but keep the types.

```
module F (X : S) : sig
  module A = (X <: S) (* X viewed through S *)
end
```

- A "filter" on the dynamic part of the module
- Also in path : `F(Y <: S).A`
- Generally useful for "soft constraints":

  `module MyKey <: Map.OrderedType = ...`
- Essential for modular implicits

Problem: Many operations on modules are expanded eagerly: `with`, `include`, `module type of`, strengthening, ...

- Lead to big signatures (Eliom has > 5Mo `.cmi`, Janestreet has build perf issues, ...)
- Loose the original information, problematic for errors and tools (documentation, notably)

Currently, when you type:

```
module type S = sig
  val x : string
  module A : T
end
module M : S with type A.t = int
```

the typechecker understands:

```
module M : sig  val x : string  module A : sig type t = int ... end  end
```

Problem: Many operations on modules are expanded eagerly: `with`, `include`, `module type of`, strengthening, ...

- Lead to big signatures (Eliom has > 5Mo `.cmi`, Janestreet has build perf issues, ...)
- Loose the original information, problematic for errors and tools (documentation, notably)

Currently, when you type:

```
module type S = sig
  val x : string
  module A : T
end
module M : S with type A.t = int
```

the typechecker understands:

```
module M : sig  val x : string  module A : sig type t = int ... end  end
```

Incrementality

Solution: Make an *incremental* calculus for modules.

- In a nutshell "memoization for module type operations".
- Similar to explicit substitutions, but on modules
- Need to be extra careful around environments!

When we typecheck "`let y = M.x`", we push only one step:

```
module M : sig
  val x : string
  module A : T with type t = int
end
```

Current status

In progress:

- Core of the specification is done
- Prototype “clean room” implementation in progress

To be done:

- Progressively add more and more features
- Prove good properties

Problem Let's write the this C&C (cool and contrived) functor:

```
module F (X : S) = struct
  type a = X.t list
  type b = X.t * int
end
```

What's the type of:

```
module A = F(struct type t = A | B end)
```

The typechecker gives us:

```
module A : sig type a type b end
```

This is not very useful.

Problem Let's write the this C&C (cool and contrived) functor:

```
module F (X : S) = struct
  type a = X.t list
  type b = X.t * int
end
```

What's the type of:

```
module A = F(struct type t = A | B end)
```

The typechecker gives us:

```
module A : sig type a type b end
```

This is not very useful.

Local bindings and inference

We want to have a *principal module type* for this.

- This type is used for inference
- It does not have to appear in user signatures

Solution: Local module types:

```
module A : let X : ... in sig
  type a = X.t list
  type b = X.t * int
end
```

We can then simplify this further down (incrementally, as before).

Local bindings and inference

We want to have a *principal module type* for this.

- This type is used for inference
- It does not have to appear in user signatures

Solution: Local module types:

```
module A : let X : ... in sig
  type a = X.t list
  type b = X.t * int
end
```

We can then simplify this further down (incrementally, as before).