An ML-Style Module System for Cross-Stage Type Abstraction in Multi-Stage Programs

FLOPS 2024

2024-05-15 @ Kumamoto, Japan

Takashi Suwa[1, 2] **Atsushi Igarashi**[1]

(1) Kyoto University (2) National Institute of Informatics

Backgrounds

Multi-stage programming (***MSP***) [Davies 1996] [Taha & Sheard 1997]

- One way to formalize languages for **metaprogramming**
- Useful as a basis of:
  - **macros** (i.e. compile-time code generation)
  - **program specialization** (i.e. runtime code generation)
- Has a notion of ***stages***
- One can write code generation **in a type-safe manner**
  - The well-typedness of generated code is statically guaranteed

Motivation: MSP with Modules

- Just as well as ordinary languages, **MSP languages should have a *module system*** [McQueen 1986]
- ***Type abstraction* by signatures is nice to have**
  - Enables us to make modules loosely-coupled

Our Work: *MetaFM*

- A module system useful for decomposing multi-stage programs into modules **without preventing type abstraction**
- Major features:
  - **Value items for different stages can be defined in a single structure** (i.e. **`struct`** … **`end`**)
  - **Covers many full-fledged module functionalities** such as:
    - (generative) higher-order functors
    - (syntactically unrestricted) projections
    - the **`with type`**-construct
    - higher-kinded types
- Formalization is based on ***F-ing Modules*** [Rossberg, Russo, & Dreyer 2014]

A Teaser for Motivating Examples

A module for handling absolute timestamps
equipped with a macro that converts a text to a timestamp

- The macro does not reveal the internal of type `Timestamp.t`
- **Type abstraction covers both compile-time and runtime**

```
module Timestamp :> sig
  type t
  val precedes : t -> t -> bool
  val advance_by_dates : t -> int -> t
  …
  ~ val generate : string -> ⟨t⟩
end = struct
  (* Implementation omitted *)
end
```

```
let our_slot_in_flops_2024 : Timestamp.t =
  ~(Timestamp.generate “2024-05-15T16:30+09:00”)
in …
```

Summary of Contributions

- Observe that value items for different stages should be able to coexist in a single structure for type abstraction
- Exemplify that such a design is achievable without hampering many realistic module features by defining ***MetaFM*** and proving its type safety
- Give ***System F$\omega^{\langle\rangle}$***, a type-safe extension of System Fω [Girard 1972] with staging features (as a target language)
- Also support ***cross-stage persistence*** [Hanada+ 2014] [Taha+ 2000]
  - A staging feature that enables us to use one common value at more than one stage

Outline

Brief introduction to multi-stage programming

- Motivating examples
- Formalization
- Discussions
  - Limitations
  - (Ongoing) future work
  - Related work
  - Conclusion

Syntax

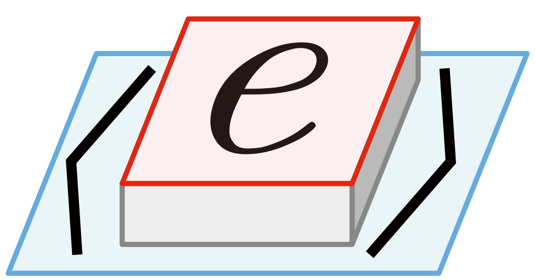
- A minimal language similar to MetaML [Taha & Sheard 1997]:

$$e ::= x \mid e\ e \mid \lambda x\,.\,e \mid \cdots \mid \langle e \rangle \mid {\sim}e$$

bracket **escape**

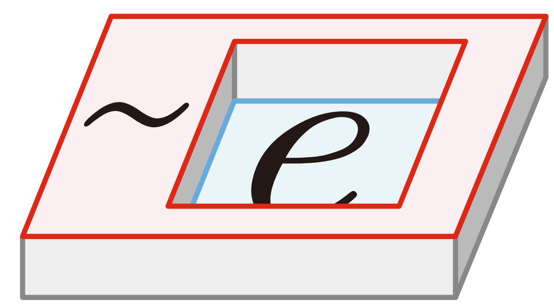
- Graphical intuition:

Bracket $\langle e \rangle$ is "convex"



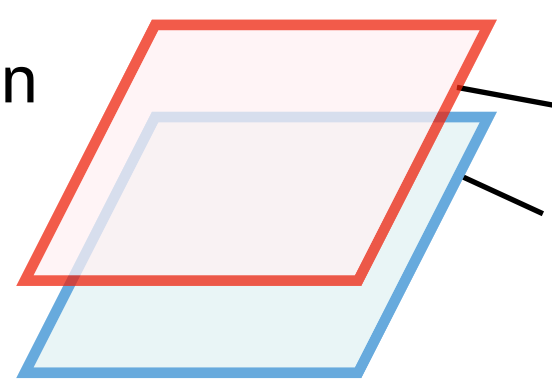
"Forms a code fragment
for the next stage"

Escape ${\sim}e$ is "concave"



"e evaluates to a code fragment
at the prev. stage and fills the hole"

Especially when
#stages = 2:



Stage 1 ≈ runtime

Stage 0 ≈ compile-time

Essence of Operational Semantics

(Especially when #stages = 2)

- Only **subexpressions at stage 0** are evaluated by the ordinary CBV β-reduction

- Escape ~ cancels bracket ⟨ ⟩ at **stage 1**
  - when a code fragment is directly inside the hole and contains no nested holes

- When the whole program reaches $\langle e \rangle$ with no holes:
  - That's the end of macro expansion
  - Then, e is used as an ordinary program

Example

- genpower:
  - Receives $m \in \mathbb{N}$ and returns code for the m-th power function

$$\textit{genpower}\ m \longrightarrow^{*} \langle \lambda a.\ \underbrace{a*a*\cdots*a}_{m}*1 \rangle$$

  - Example use:

```
let cubic = ~(genpower 3) in ⋯
```

- cf. the usual non-staged power function

```
let cubic = power 3 in ⋯
```

  - cubic incurs recursive calls at runtime

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

genpower 2 $\longrightarrow^{*}$ ⟨λa. ~(aux 2 ⟨a⟩)⟩

Generates a fresh symbol
for hygienicity
(not mentioned henceforth)

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
```

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
```

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~⟨1⟩⟩⟩⟩
```

Example Reduction

```
let rec aux n s =
  if n <= 0 then ⟨1⟩ else
    ⟨~s * ~(aux (n - 1) s)⟩

let genpower n = ⟨λx. ~(aux n ⟨x⟩)⟩
```

```
genpower 2  →*  ⟨λa. ~(aux 2 ⟨a⟩)⟩
            →*  ⟨λa. ~⟨~⟨a⟩ * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~(aux 1 ⟨a⟩)⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~(aux 0 ⟨a⟩)⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * ~⟨1⟩⟩⟩⟩
            →*  ⟨λa. ~⟨a * ~⟨a * 1⟩⟩⟩
            →*  ⟨λa. a * (a * 1)⟩
```

Minimal Type System for Staging

[Taha & Sheard 1997]

- ***Code types*** are added: $\tau ::= \cdots \mid \langle \tau \rangle$
  - "The type for code fragments that will be expressions of type τ at the next stage"
  - e.g. `genpower` : $\mathbf{int} \to \langle \mathbf{int} \to \mathbf{int} \rangle$

- Especially prevents situations where:
  - finally produced code contains an unbound variable

```
⟨λx. y⟩
```

  - generated code is ill-typed

```
(λt. ⟨~t * 3⟩) ⟨true⟩  →*  ⟨true * 3⟩
```

Outline

- Brief introduction to multi-stage programming
- **Motivating examples**
- Formalization
- Discussions
  - Limitations
  - (Ongoing) future work
  - Related work
  - Conclusion

Example Use of Our Module System

A module for handling absolute timestamps:

```
module Timestamp :> sig
  type t
  val precedes : t -> t -> bool
  val advance_by_dates : t -> int -> t
  …
end = struct
  type t = int (* Internally in Unix time *)
  val precedes ts1 ts2 = ts1 < ts2
  val advance_by_dates ts dates =
  …
end
```

It would be nice if we can use a macro like the following:

```
let our_slot_in_flops_2024 : Timestamp.t =
  ~(Timestamp.generate “2024-05-15T16:30+09:00”)
in …
```

Example Use of Our Module System

```
module Timestamp :> sig
  type t
  val precedes : t -> t -> bool
  val advance_by_dates : t -> int -> t
  …
  ~ val generate : string -> ⟨t⟩
end = struct
  type t = int (* Internally in Unix time *)
  val precedes ts1 ts2 = ts1 < ts2
  val advance_by_dates ts dates =
  …
  ~ val generate s =
      match parse_datetime s with
      | None    -> failwith “invalid datetime”
      | Some ts -> lift ts

end
```

```
let our_slot_in_flops_2024 : Timestamp.t =
  ~ (Timestamp.generate “2024-05-15T16:30+09:00”)
in …
```

Example Use of Our Module System

```
module Timestamp :> sig
  type t
  val precedes : t -> t -> bool
  val advance_by_dates : t -> int -> t
  …
  ~val generate : string -> ⟨t⟩
end = struct
  type t = int (* Internally in Unix time *)
  val precedes ts1 ts2 = ts1 < ts2
  val advance_by_dates ts dates =
  …
  ~val generate s =
     match parse_datetime s with
     | None     -> failwith "invalid datetime"
     | Some ts -> lift ts

end
```

Macros do not expose the internal representation of type `Timestamp.t` as well as ordinary values do not

```
let our_slot_in_flops_2024 : Timestamp.t =
  ~(Timestamp.generate "2024-05-15T16:30+09:00")
in …
```

Example Use of Our Module System

```
module Timestamp :> sig
  type t
  val precedes : t -> t -> bool
  val advance_by_dates : t -> int -> t
  …
  ~ val generate : string -> ⟨t⟩
end = struct
  type t = int (* Internally in Unix time *)
  val precedes ts1 ts2 = ts1 < ts2
  val advance_by_dates ts dates =
  …
  ~ val generate s =
      match parse_datetime s with
      | None    -> failwith “invalid datetime”
      | Some ts -> lift ts
end
```

Macros do not expose the internal representation of type `Timestamp.t` as well as ordinary values do not

Lifts a value to the next stage

e.g. `lift 5` $\longrightarrow^{*}$ ⟨5⟩

```
let our_slot_in_flops_2024 : Timestamp.t =
  ~(Timestamp.generate “2024-05-15T16:30+09:00”)
in …
```

An Example involving Functors

- A macro offered by `MakeMap` (= OCaml's `Map.Make`) that converts a list of key-value pairs to a map beforehand

```
module StringMap = MakeMap(String)

let month_abbrev_to_int (s : string) : option int =
  StringMap.find_opt s
    ~(StringMap.generate [(“Jan”, 1), …, (“Dec”, 12)]
```

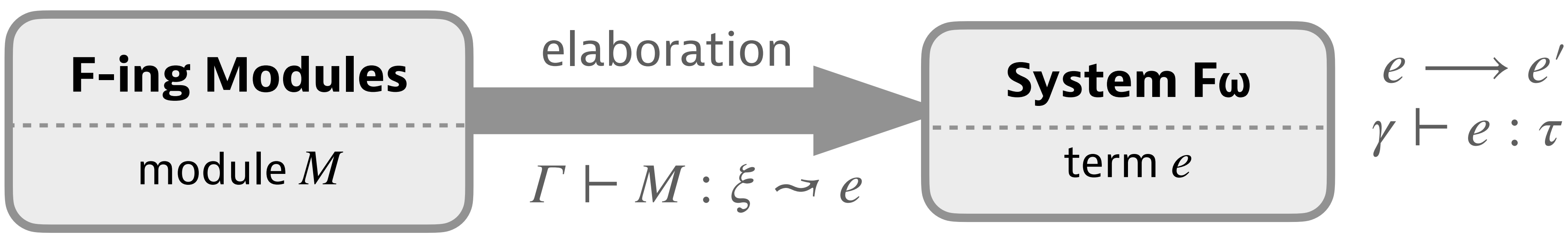
```
module MakeMap :> (Key : Ord) -> sig
  type t :: * -> *
  val empty : ∀α. t α
  val find_opt : ∀α. Key.t -> t α -> option α
  …
  ~val generate : ∀α. list (Key.t × α) -> ⟨t α⟩
end = fun(Key : Ord) -> struct
  type t α = Leaf | Node of … (* Balanced binary tree *)
  val empty = Leaf
  val find_opt key map = …
  …
  ~val generate kvs = …
end
```

Outline

- Brief introduction to multi-stage programming
- Motivating examples
- **Formalization**
- Discussions
  - Limitations
  - (Ongoing) future work
  - Related work
  - Conclusion

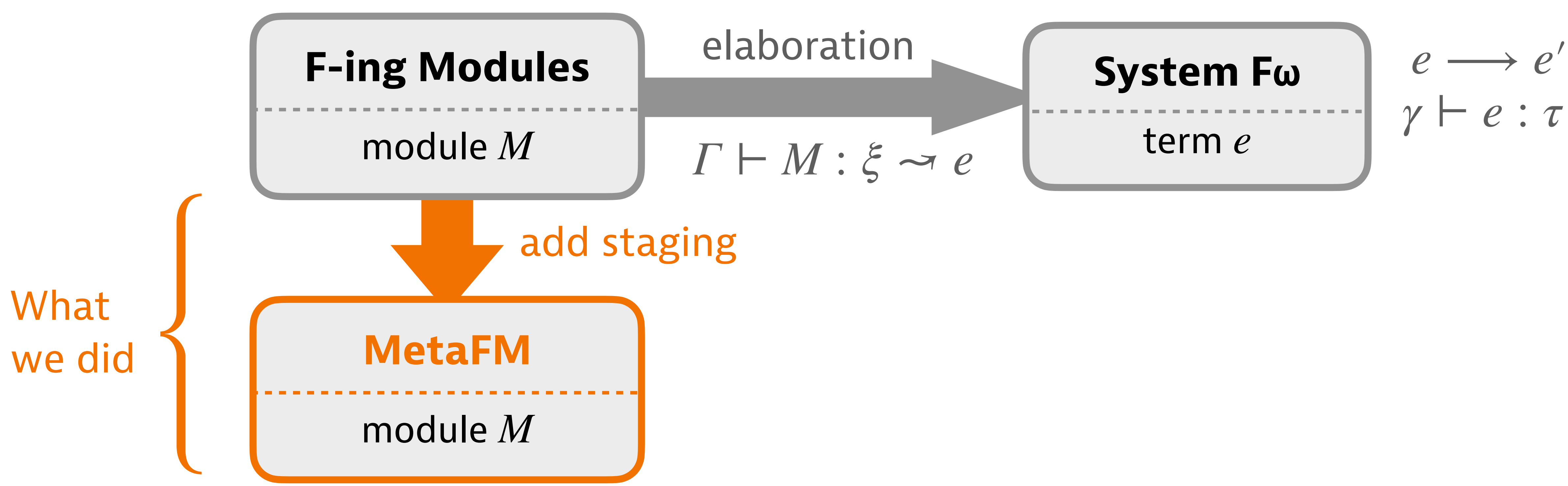
How to Define Semantics & Type Safety

- cf. F-ing Modules [Rossberg, Russo, & Dreyer 2014]
  - Uses an ***elaboration*** technique to define semantics
    - Type-directed conversion of modules into System Fω terms
  - Proves type safety in two steps:
    1. Any elaborated term is well-typed under System Fω
    2. System Fω [Girard 1972] fulfills Preservation & Progress



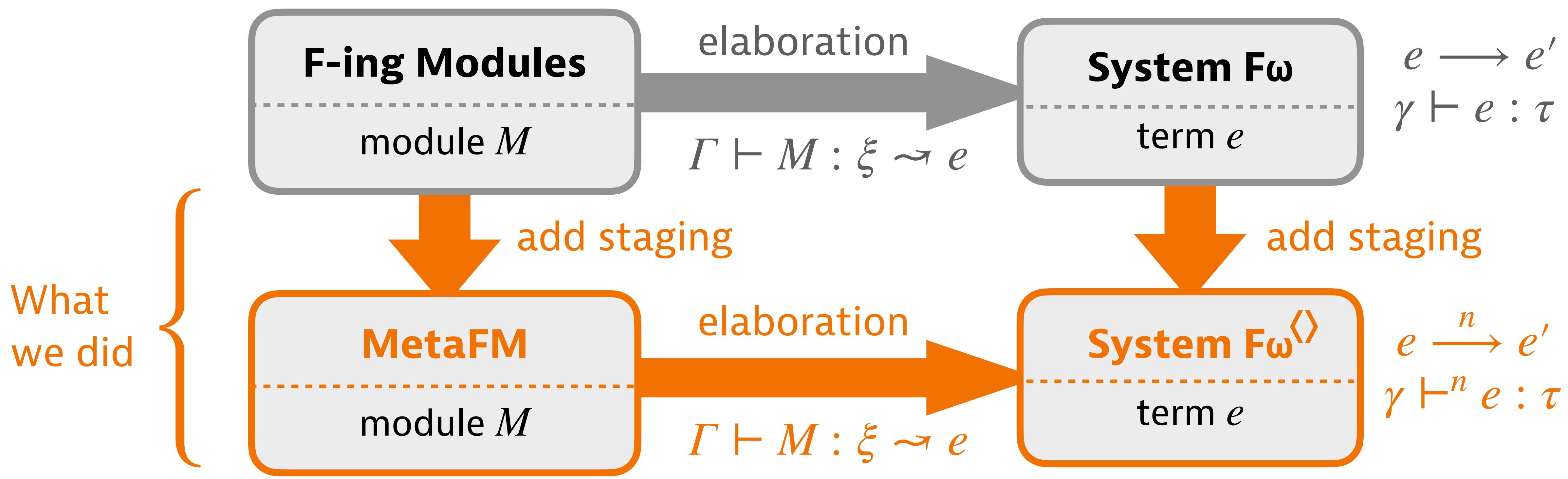
How to Define Semantics & Type Safety

- cf. F-ing Modules [Rossberg, Russo, & Dreyer 2014]
  - Uses an ***elaboration*** technique to define semantics
    - Type-directed conversion of modules into System Fω terms
  - Proves type safety in two steps:
    1. Any elaborated term is well-typed under System Fω
    2. System Fω [Girard 1972] fulfills Preservation & Progress

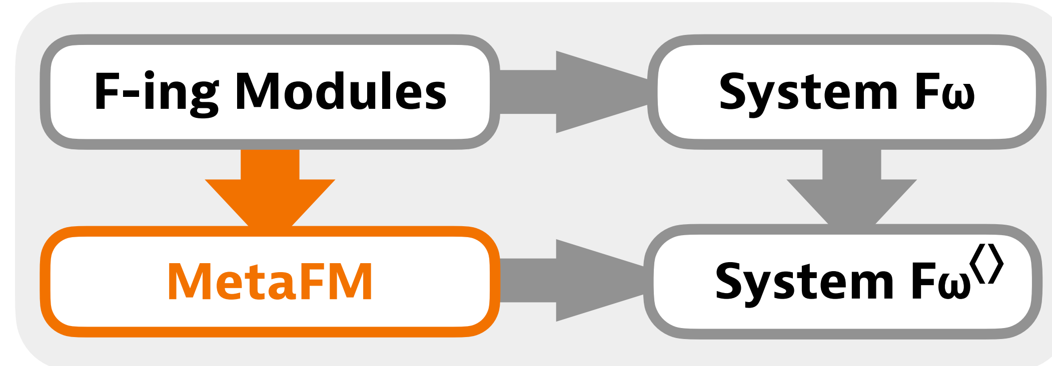


How to Define Semantics & Type Safety

- Our work:
  - Also proves type safety in two steps:
    1. Any elaborated term is well-typed under ***System Fω***$^{\langle\rangle}$
    2. ***System Fω***$^{\langle\rangle}$ fulfills Preservation & Progress



Source Syntax



bindings

$$
\begin{aligned}
B ::= {} & \mathbf{val}^n\ X = E \\
\mid {} & \mathbf{type}\ X = T \\
\mid {} & \mathbf{module}\ X = M \\
\mid {} & \mathbf{include}\ M
\end{aligned}
$$

declarations

$$
\begin{aligned}
D ::= {} & \mathbf{val}^n\ X : T \\
\mid {} & \mathbf{type}\ X :: K \\
\mid {} & \mathbf{module}\ X : S \\
\mid {} & \mathbf{include}\ S
\end{aligned}
$$

higher-kinded

modules

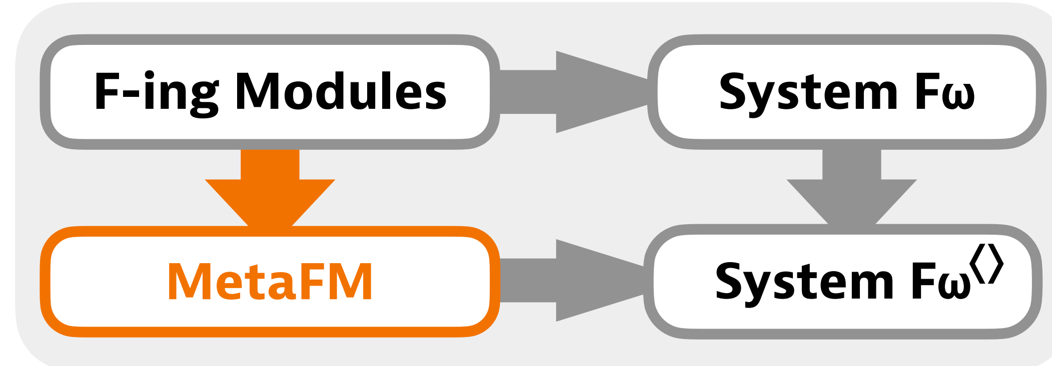
$$
\begin{aligned}
M ::= {} & X \mid M.X && \text{var. \& projection} \\
\mid {} & \mathbf{struct}\ \overline{B}\ \mathbf{end} && \text{structures} \\
\mid {} & \mathbf{fun}(X : S) \to M && \text{functor abs./app.} \\
\mid {} & X\ X && \\
\mid {} & X :> S && \text{sealing}
\end{aligned}
$$

signatures

$$
\begin{aligned}
S ::= {} & \mathbf{sig}\ \overline{D}\ \mathbf{end} \\
\mid {} & (X : S) \to S \\
\mid {} & S\ \mathbf{with\ type}\ \overline{X} = T
\end{aligned}
$$

- Almost the same as ***F-ing Modules*** [Rossberg+ 2014] except for $\mathbf{val}^n\ X$
- $\sim\mathbf{val}$ and $\mathbf{val}$ were shorthand for $\mathbf{val}^0$ and $\mathbf{val}^1$

Source Syntax



n specifies for which stage the value X is defined

bindings

$$
\begin{aligned}
B ::= {} & \mathbf{val}^n\ X = E \\
\mid {} & \mathbf{type}\ X = T \\
\mid {} & \mathbf{module}\ X = M \\
\mid {} & \mathbf{include}\ M
\end{aligned}
$$

declarations

$$
\begin{aligned}
D ::= {} & \mathbf{val}^n\ X : T \\
\mid {} & \mathbf{type}\ X :: K \\
\mid {} & \mathbf{module}\ X : S \\
\mid {} & \mathbf{include}\ S
\end{aligned}
$$

higher-kinded

modules

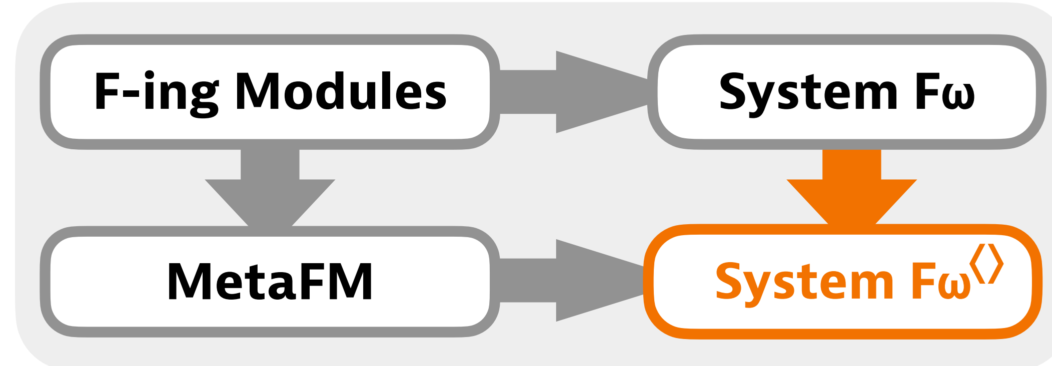
$$
\begin{aligned}
M ::= {} & X \mid M \,.\, X && \text{var. \& projection} \\
\mid {} & \mathbf{struct}\ \overline{B}\ \mathbf{end} && \text{structures} \\
\mid {} & \mathbf{fun}(X : S) \to M && \\
\mid {} & X\ X && \text{functor abs./app.} \\
\mid {} & X :> S && \text{sealing}
\end{aligned}
$$

signatures

$$
\begin{aligned}
S ::= {} & \mathbf{sig}\ \overline{D}\ \mathbf{end} \\
\mid {} & (X : S) \to S \\
\mid {} & S\ \mathbf{with\ type}\ \overline{X} = T
\end{aligned}
$$

- Almost the same as ***F-ing Modules*** [Rossberg+ 2014] except for $\mathbf{val}^n\ X$
- $\sim\mathbf{val}$ and $\mathbf{val}$ were shorthand for $\mathbf{val}^0$ and $\mathbf{val}^1$

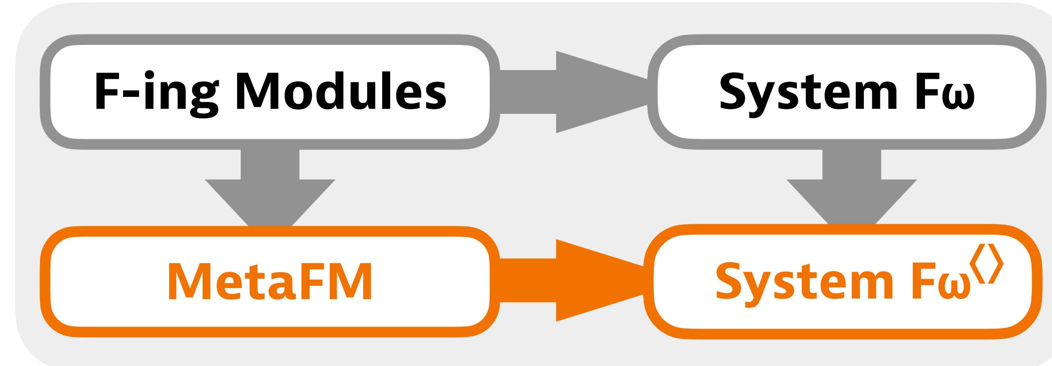
Target Language: *System F$\omega^{\langle\rangle}$*



- An extension of System Fω [Girard 1972] with staging constructs
- Allows existentials only at stage 0
  - This suffices for the elaboration of MetaFM
  - Has no difficulty in mixing existentials and staging

$$
\begin{array}{rrcl}
\text{terms} & e & ::= & \cdots \mid \textbf{pack}\ (\tau, e)\ \textbf{as}\ \exists\alpha\,.\,\tau \mid \cdots \mid \overset{\textit{bracket}}{\langle e\rangle} \mid \overset{\textit{escape}}{\sim e} \\
\text{higher-kinded types} & \tau & ::= & \alpha \mid \tau\ \tau \mid \exists\alpha :: \kappa\,.\,\tau \mid \cdots \mid \underset{\textit{code types}}{\langle\tau\rangle} \\
\text{kinds} & \kappa & ::= & \bullet \mid \kappa \to \kappa
\end{array}
$$

Essence of Elaboration



- Leaving types out of account, elaboration is simply like:

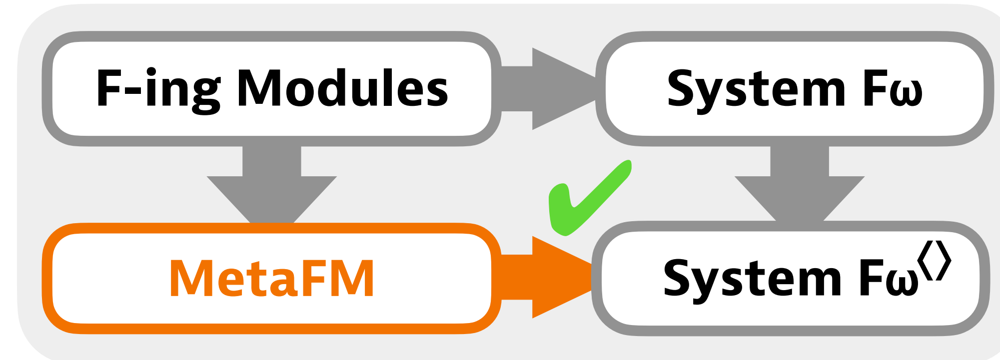
$$\textbf{val}^{n}\ X = E \quad \Longrightarrow \quad \textbf{let}\ X = \underbrace{\langle \cdots \langle}_{n}\ E\ \underbrace{\rangle \cdots \rangle}_{n}$$

$$M\,.\,X \ \ \text{(at stage } n\text{)} \quad \Longrightarrow \quad \underbrace{\sim \cdots \sim}_{n}(M\,.\,X)$$

- Though somewhat naïve in that it changes binding time, this elaboration at least fulfills type safety
  - Related issues will be discussed later

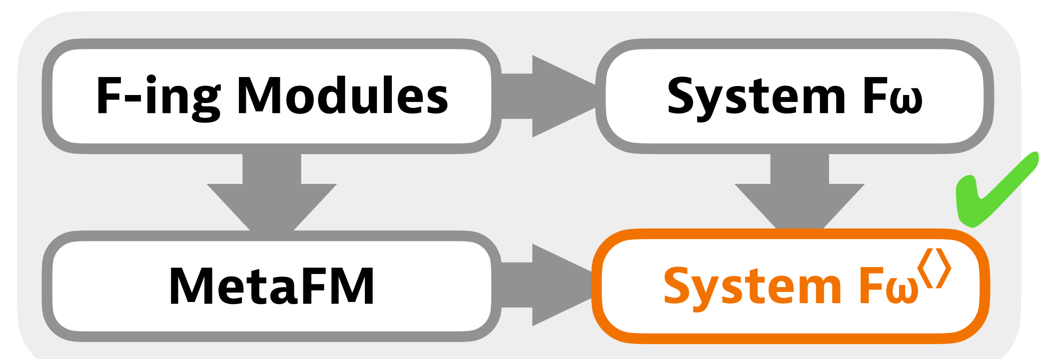
Correctness of MetaFM

1. Any elaborated term is well-typed:



Theorem
If $\Gamma \vdash M : \xi \rightsquigarrow e$, then $\lfloor \Gamma \rfloor \vdash^{0} e : \lfloor \xi \rfloor$.

2. Target type safety:



Theorem (Preservation).
If $\gamma \vdash^{n} e : \tau$ and $e \xrightarrow{n} e'$, then $\gamma \vdash^{n} e' : \tau$.

Theorem (Progress).
If $\vdash^{\geq 1} \gamma$ and $\gamma \vdash^{n} e : \tau$, then
e is a value at stage n,
or there exists e' such that $e \xrightarrow{n} e'$.

Extension with Cross-Stage Persistence

- ***Cross-stage persistence*** (***CSP***) [Taha & Sheard 2000]
  - A multi-stage feature that enables us to **use one common value at more than one stage**
  - Useful, e.g., when one wants to use basic functions (such as `(+)` or `List.map`) at both compile-time and runtime

Extension with Cross-Stage Persistence

- ***Cross-stage persistence*** (***CSP***) [Taha & Sheard 2000]
  - A multi-stage feature that enables us to **use one common value at more than one stage**
  - Useful, e.g., when one wants to use basic functions (such as `(+)` or `List.map`) at both compile-time and runtime

X **will be bound as a value usable at any stage** $n' \ (\geq n)$

- Formalization:
  - Add a binding syntax: $B \ ::= \ \mathbf{val}^n \ X = E \ \mid \ \mathbf{val}^{\geq n} \ X = E \ \mid \cdots$
  - Extend both source & target type systems with ***stage var.***
    - A limited version of ***env. classifiers*** [Taha & Nielsen 2003] or ***transition var.*** [Tsukada+ 2009] [Hanada+ 2014]

- … See our paper for detail!

Outline

- Brief introduction to multi-stage programming
- Motivating examples
- Formalization
- **Discussions**
  - **Limitations**
  - **(Ongoing) future work**
  - **Related work**
  - **Conclusion**

Limitations

- Does not support the ***Run primitive*** [Taha+ 1997]
  - Example: **`run`** `(genpower 3) 5` $\longrightarrow^{*}$ `125`
  - Can perhaps be overcome by some orthogonal methods
- Cannot extend with **first-class modules**
  - Currently regards all modules as stage-0 stuff
- Cannot accommodate features with **effects** such as **mutable refs**
  - Because of the binding-time change

Issues on Mutable Refs

Stage-1 expressions containing mutable refs are converted to target expressions that have unintended behavior

```
module M = struct
  val x = ref 42

  val main () =
    x := 57;
    print !x
end
```

elaboration

```
let M' =
  let x' = ⟨ref 42⟩ in
  let main' =
    ⟨λ().
      ~x' := 57;

      ~print' !~x' ⟩
  in
  { x = x' ; main = main' }
```

`M.main ()` is expected to print 57

Will generate

```
λ().
  (ref 42) := 57;
  (λn. ⋯) !(ref 42)
```

which prints 42

Recall: elaboration is like:

$$\mathbf{val}^n\ X = E \quad \Longrightarrow \quad \mathbf{let}\ X = \underbrace{\langle \cdots \langle}_{n}\ E\ \underbrace{\rangle \cdots \rangle}_{n}$$

$$M\,.\,X \quad \Longrightarrow \quad \underbrace{\sim \cdots \sim}_{n}(M\,.\,X)$$

Ongoing Work: Refine Elaboration

- We can probably define better elaboration rules by using ***static interpretation*** [Elsman 1999] [Bochao+ 2010]
  - **Converts module structures into a flat list of bindings of the form** $\mathrm{val}^n\ x = e$ (with functor applications resolved)

```
module M = struct
  val x = ref 42

  val main () =
    x := 57;
    print !x
end
```

elaboration

```
(val¹ precedes' = λts1. λts2. ⋯);
(val⁰ generate' = λts. ⋯);
⋯
(val¹ x' = ref 42);
(val¹ main' = λ().
  x' := 57;
  print' !x')
```

  - We implemented promising elaboration rules for ***SATySFI*** [Suwa 2018] and observed that they work fine with mutable refs
    - SATySFI: An ML-like statically typed language for typesetting documents
  - ***Let-insertion*** [Danvy & Fillinski 1990] [Sato+ 2020] could also be effective, but it may complicate semantics and its correctness

Related Work 1: Staging Modules

- Staging beyond terms [Inoue, Kiselyov, & Kameyama 2016]
- Program generation for ML modules [Watanabe & Kameyama 2018]
- Module generation without regret [Sato, Kameyama, & Watanabe 2020]

	The studies above	**MetaFM (ours)**
Basic purpose	Elimination of overheads caused by functors by using staging	Provide a realistic module system for MSP, especially from the viewpoint of type abstraction
Language design	Staging whole module expressions • Seems ineffective for the purpose of type abstraction	Staging each item individually

Related Work 2: MacoCaml [Xie, White, Nicole, & Yallop 2023]

	MacoCaml	MetaFM (ours)
Basic purpose	Extend OCaml with type-safe, composable macros	Provide MSP languages with full-blown module features, especially with type abs.
Formalization of semantics	Given directly on source syntactic entities	Given through elaboration to System F$\omega^{\langle\rangle}$
Functors	✖	✔ **Supported**
Type abs.	✖	✔ **Supported**
Avoidance problem [Lillibridge 1997] [Crary 2020]	🙁 Extending with proj. $M.X$ and type abs. by $X :> S$ may well cause this issue	✔ **Free from this concern** thanks to the elaboration
Eval. order	✔ Intuitive • Supports mutable refs	🙁 Currently causes a gap between users' intuition and actual behavior of target terms • Probably remedied by ongoing work
CSP	✔ By $\mathbf{import}^{\downarrow}$	✔ By $\mathbf{val}^{\geq n}\ X = E$
Run prim.	✖	✖

Conclusion

- ***MetaFM***: a module system that enables us to decompose ***multi-stage*** programs into modules without preventing type abstraction

- Supports many important features:
  - Advanced module operations
    - (generative) higher-order functors, projection, higher-kinded types, etc.
  - ***Cross-stage persistence*** [Taha+ 2000] by the form $\mathbf{val}^{\geq n}\ X = E$

- Has limitations that should be remedied by future work
  - Cannot extend with effectful computation
    - Probably overcome by ***static-interpretation***-based elaboration [Elsman 1999]
  - Cannot handle first-class modules

References

1. L. Bochao and A. Ohori. A flattening strategy for SML module compilation and its implementation. *Information and Media Technologies*, **5**(1), 2010.
2. K. Crary. A focused solution to the avoidance problem. *Journal of Functional Programming*, 2020.
3. O. Danvy and A. Filinski. Abstracting control. In *Proc. of LFP*, 1990.
4. M. Elsman. Static interpretation of modules. In *Proc. of ICFP*, 1999.
5. M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea. Static interpretation of higher-order modules in Futhark: functional GPU programming in the large. In *Proc. of ICFP*, 2018.
6. J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII, 1972.
7. Y. Hanada and A. Igarashi. On cross-stage persistence in multi-stage programming. In *Proc. of FLOPS*, 2014.
8. J. Inoue, O. Kiselyov, and Y. Kameyama. Staging beyond terms: prospects and challenges. In *Proc. of PEPM*, 2016.
9. M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. Ph.D. thesis, Carnegie Mellon University, 1997.
10. A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, **24**(5), 2014.
11. Y. Sato, Y. Kameyama, and T. Watanabe. Module generation without regret. In *Proc. of PEPM*, 2020.
12. W. Taha and M. Nielsen. Environment classifiers. In *Proc. of POPL*, 2003.
13. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. of PEPM*, 1997.
14. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, **248**(1-2), 2000.
15. T. Tsukada and A. Igarashi. A logical foundation for environment classifiers. In *Proc. of TLCA*. volume 5608 of *Lecture Notes in Computer Science*, 2009.
16. T. Watanabe and Y. Kameyama. Program generation for ML modules (short paper). In *Proc. of PEPM*, 2018.
17. N. Xie, L. White, O. Nicole, and J. Yallop. MacoCaml: staging composable and compilable macros. In *Proc. of ICFP*, 2023.

Appendix A:
Auxiliary Materials

Syntax Sugars [Rossberg, Russo, & Dreyer 2014]

- Transparent declarations of types:

$$\textbf{type}\ X = T \ :=\ \textbf{include}\ (\textbf{struct type}\ X :: K\ \textbf{end with type}\ X = T)$$

  - where K should be inferred from T

- Local bindings by projection:

$$\textbf{let}\ \overline{B}\ \textbf{in}\ M \ :=\ (\textbf{struct}\ \overline{B};\ \textbf{module}\ X = M\ \textbf{end})\,.\,X$$
$$\textbf{let}\ \overline{B}\ \textbf{in}\ E \ :=\ (\textbf{struct}\ \overline{B};\ \textbf{val}\ X = E\ \textbf{end})\,.\,X$$

  - where X is fresh

- Functor app. and sealing generalized for arbitrary modules:

$$M_1\ M_2 \ :=\ \textbf{let module}\ X_1 = M_1;\ \textbf{module}\ X_2 = M_2\ \textbf{in}\ X_1\ X_2$$
$$M :> S \ :=\ \textbf{let module}\ X = M\ \textbf{in}\ X :> S$$

  - where X_1, X_2, and X are fresh

Avoidance Problem [Lillibridge 1997] [Crary 2020]

- You cannot simply reject entities that refer to local types:

```
let Local =
  … :> sig
    type t
    val x : t
  end
in Local.x
```

✖ Rejected

Type Local.t is escaping its scope

```
let Local =
  … :> sig
    type t = int
    val x : t
  end
in Local.x
```

✔ OK

Assigned type Local.t (= int)

- But, for a module depending on some local types, in general **there's no principal signature that avoids mentioning the local types escaping the scope**

```
module M =
  let type foo = Foo in
  … :> sig
    type dummy α = foo
    type bar = Bar of foo
    val x : dummy int
    val y : dummy bool
  end
```

– Both `M.(Bar x)` and `M.(Bar y)` should type-check, but no signature for `M` that avoids mentioning `foo` achieves it (without special mechanisms)

Staging Modules isn’t Effective

```
module Timestamp :> sig
  type t
  val make : int -> t
  …
end = struct
  type t = int (* Internally in Unix time *)
  val make ts = ts
  …
end
```

We have to make a backdoor that exposes internal details

```
module GenTimestamp :> sig
  val generate : string -> ⟨Timestamp.t⟩
end = struct
  val generate s =
    match parse_datetime s with
    | None    -> failwith “invalid datetime”
    | Some ts -> ⟨Timestamp.make ~(lift ts)⟩
end
```

For type abstraction, we have to leave at least one fun. app. at stage 1

Appendix B:
Basic Elaboration Rules

Example of Elaboration

```
sig
  type t :: *
  val precedes : t -> t -> bool
  ~val generate : string -> ⟨t⟩
  …
end
```

$$
\begin{aligned}
&\exists \beta :: \bullet\,.\,\{\\
&\quad l_{\mathtt{t}} \mapsto \llparenthesis = \beta :: \bullet \rrparenthesis,\\
&\quad l_{\mathtt{precedes}} \mapsto \llparenthesis \beta \to \beta \to \mathrm{bool} \rrparenthesis^{1},\\
&\quad l_{\mathtt{generate}} \mapsto \llparenthesis \mathrm{string} \to \langle \beta \rangle \rrparenthesis^{0},\\
&\quad \ldots\}
\end{aligned}
$$

```
(Key : sig
  type t :: *
  val≥0 compare : t -> t -> int
end) -> sig
  type t :: * -> *
  val empty : ∀α. t α
  val find_opt :
    ∀α. Key.t -> t α -> option α
  ~val generate :
    ∀α. list (Key.t × α) -> ⟨t α⟩
  …
end
```

$$
\begin{aligned}
&\forall \chi :: \bullet\,.\,\{\\
&\quad l_{\mathtt{t}} \mapsto \llparenthesis = \chi :: \bullet \rrparenthesis,\\
&\quad l_{\mathtt{compare}} \mapsto \llparenthesis \chi \to \chi \to \mathrm{int} \rrparenthesis^{\geq 0}\\
&\} \to \exists \beta :: \bullet \to \bullet\,.\,\{\\
&\quad l_{\mathtt{t}} \mapsto \llparenthesis = \beta :: \bullet \to \bullet \rrparenthesis,\\
&\quad l_{\mathtt{empty}} \mapsto \llparenthesis \forall \alpha :: \bullet\,.\, \beta\ \alpha \rrparenthesis^{1},\\
&\quad l_{\mathtt{find\_opt}} \mapsto \llparenthesis \forall \alpha :: \bullet\,.\, \chi \to \beta\ \alpha \to \mathrm{option}\ \alpha \rrparenthesis^{1},\\
&\quad l_{\mathtt{generate}} \mapsto \llparenthesis \forall \alpha :: \bullet\,.\, \mathrm{list}\ (\chi \times \alpha) \to \langle \beta\ \alpha \rangle \rrparenthesis^{0},\\
&\quad \ldots\}
\end{aligned}
$$

Semantic Signatures & Target Types

- Internal representation of signatures used in type-checking

concrete sig.	$\Sigma ::= \boxed{(\!|\,\tau\,|\!)^{n}}$	**value items for stage n**
	$\mid\ (\!| = \tau :: \kappa |\!)$	type items
	$\mid\ \{\overline{l_X : \Sigma}\}$	(internal) structure sig.
	$\mid\ \forall \overline{\alpha :: \kappa}\,.\ \Sigma \to \xi$	(internal) functor sig.
abstract sig.	$\xi ::= \exists \overline{\alpha :: \kappa}\,.\ \Sigma$	

- Updates from F-ing Modules [Rossberg+ 2014] and Fω types:
  - **The stage number superscript n of $(\!|\,\tau\,|\!)^{n}$**
  - Code types: $\tau \ ::= \ \alpha \mid \tau\ \tau \mid \cdots \mid \boxed{\langle \tau \rangle}$

Signature Elaboration

$$\Gamma ::= \cdot \mid \Gamma, X : \Sigma \mid \Gamma, \alpha :: \kappa$$

$\boxed{\Gamma \vdash S \rightsquigarrow \xi}$

"Under type env. Γ, sig. S is interpreted as abstract sig. ξ."

$$\frac{\Gamma \vdash \boldsymbol{D} \rightsquigarrow \exists \boldsymbol{b} . R}{\Gamma \vdash \textbf{sig}\ \boldsymbol{D}\ \textbf{end} \rightsquigarrow \exists \boldsymbol{b} . \{R\}}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \boldsymbol{b} . \Sigma_1 \qquad \Gamma, \boldsymbol{b}, X : \Sigma_1 \vdash S_2 \rightsquigarrow \xi_2}{\Gamma \vdash (X : S_1) \to S_2 \rightsquigarrow \exists \epsilon . (\forall \boldsymbol{b} . \Sigma_1 \to \xi_2)}$$

$\boxed{\Gamma \vdash \boldsymbol{D} \rightsquigarrow \exists \boldsymbol{b} . R}$

$$\frac{}{\Gamma \vdash \epsilon \rightsquigarrow \exists \epsilon . \varnothing}$$

$$\frac{\Gamma \vdash D_1 \rightsquigarrow \exists \boldsymbol{b}_1 . R_1 \qquad \operatorname{dom} \boldsymbol{b}_1 \cap \operatorname{tv} \Gamma = \varnothing \qquad \Gamma, \boldsymbol{b}_1, R_1 \vdash \boldsymbol{D}_2 \rightsquigarrow \exists \boldsymbol{b}_2 . R_2 \qquad \operatorname{dom} \boldsymbol{b}_2 \cap \operatorname{dom} \boldsymbol{b}_1 = \varnothing}{\Gamma \vdash D_1 \cdot \boldsymbol{D}_2 \rightsquigarrow \exists \boldsymbol{b}_1 \boldsymbol{b}_2 . R_1 \uplus R_2}$$

$\boxed{\Gamma \vdash D \rightsquigarrow \exists \boldsymbol{b} . R}$

Introduces type var.

$$\frac{\Gamma \vdash K \rightsquigarrow \kappa}{\Gamma \vdash \textbf{type}\ X :: K \rightsquigarrow \exists \alpha :: \kappa . \{l_X \mapsto \llparenthesis = \alpha :: \kappa \rrparenthesis\}}$$

$$\frac{\Gamma \vdash T :: \bullet \rightsquigarrow \tau}{\Gamma \vdash \textbf{val}^n\ X : T \rightsquigarrow \exists \epsilon . \{l_X \mapsto \llparenthesis \tau \rrparenthesis^n\}}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \boldsymbol{b} . \Sigma}{\Gamma \vdash \textbf{module}\ X : S \rightsquigarrow \exists \boldsymbol{b} . \{l_X \mapsto \Sigma\}}$$

Elaboration Rules

$$\boxed{\Gamma \vdash M : \xi \rightsquigarrow e}$$

"Under type env. Γ, module expr. M is assigned abstract sig. ξ and converted to term e."

$$\frac{\Gamma \vdash \boldsymbol{B} : \exists \boldsymbol{b} \,.\, R \rightsquigarrow e}{\Gamma \vdash \textbf{struct}\ \boldsymbol{B}\ \textbf{end} : \exists \boldsymbol{b} \,.\, \{R\} \rightsquigarrow e}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \boldsymbol{b} \,.\, \Sigma_1 \qquad \Gamma, \boldsymbol{b}, X : \Sigma_1 \vdash M_2 : \xi_2 \rightsquigarrow e_2}{\Gamma \vdash \textbf{fun}(X : S_1) \rightarrow M_2 : (\forall \boldsymbol{b} \,.\, \Sigma_1 \rightarrow \xi_2) \rightsquigarrow (\Lambda \boldsymbol{b} \,.\, \lambda X_1 \,.\, e_2)}$$

Subtyping produces embodied types and an injection fun.

$$\frac{\Gamma(X_1) = \forall \boldsymbol{b} \,.\, \Sigma \rightarrow \xi \qquad \Gamma(X_2) = \Sigma_2 \qquad \Gamma \vdash \Sigma_2 \leqslant \exists \boldsymbol{b} \,.\, \Sigma \uparrow \boldsymbol{\tau} \rightsquigarrow f}{\Gamma \vdash X_1\ X_2 : [\boldsymbol{\tau}/\boldsymbol{b}]\xi \rightsquigarrow X_1\ \boldsymbol{\tau}\ (f\ X_2)}$$

$$\boxed{\Gamma \vdash \boldsymbol{B} : \exists \boldsymbol{b} \,.\, R \rightsquigarrow e}$$

(nil and cons; elaboration is complicated due to intro./elim. of ∃)

$$\frac{\begin{array}{c}\Gamma \vdash B_1 : \exists \boldsymbol{b}_1 .\ R_1 \rightsquigarrow e_1 \qquad \mathrm{dom}\, \boldsymbol{b}_1 \cap \mathrm{domtv}\, \Gamma = \varnothing \\ \Gamma, \boldsymbol{b}_1, R_1 \vdash \boldsymbol{B}_2 : \exists \boldsymbol{b}_2 .\ R_2 \rightsquigarrow e_2 \qquad \mathrm{dom}\, \boldsymbol{b}_2 \cap \mathrm{dom}\, \boldsymbol{b}_1 = \varnothing \\ \hat{r}_1 = \{l_X \mapsto x_1\#l_X \mid l_X \in \mathrm{dom}\, R_1 \setminus \mathrm{dom}\, R_2\} \qquad \boldsymbol{b} = \boldsymbol{b}_1 \cdot \boldsymbol{b}_2 \\ \hat{r}_2 = \{l_X \mapsto x_2\#l_X \mid l_X \in \mathrm{dom}\, R_2\} \qquad R = R_1 + R_2\end{array}}{\begin{array}{l}\Gamma \vdash B_1 \cdot \boldsymbol{B}_2 : \exists \boldsymbol{b} .\ R \rightsquigarrow \textbf{unpack}\ (\boldsymbol{b}_1, x_1 : \lfloor \{\!| R_1 |\!\} \rfloor) = e_1\ \textbf{in} \\ \qquad \textbf{unpack}\ (\boldsymbol{b}_2, x_2 : \lfloor \{\!| R_2 |\!\} \rfloor) = \\ \qquad\quad \textbf{let}\ \{X : \lfloor \Sigma \rfloor = x_1\#l_X \mid (l_X \mapsto \Sigma) \in R_1\}\ \textbf{in}\ e_2\ \textbf{in} \\ \qquad \textbf{pack}\ (\boldsymbol{b}, \{\hat{r}_1 \uplus \hat{r}_2\})\ \textbf{as}\ \lfloor \exists \boldsymbol{b} .\ \{\!| R |\!\} \rfloor\end{array}}\ \textsc{Bs-Cons}$$

Elaboration Rules

$$\boxed{\Gamma \vdash B : \exists \boldsymbol{b} \,.\, R \leadsto e}$$

$$\frac{\Gamma \vdash M : \exists \boldsymbol{b} \,.\, \Sigma \leadsto e}{\Gamma \vdash \textbf{module}\ X = M : \exists \boldsymbol{b} \,.\, \{l_X \mapsto \Sigma\} \leadsto \{l_X \mapsto e\}}$$

$$\frac{\Gamma \vdash^n E : \tau \leadsto e}{\Gamma \vdash \textbf{val}^n\ X = E : \exists \epsilon \,.\, \{l_X \mapsto \llparenthesis \tau \rrparenthesis^n\} \leadsto \{l_X \mapsto \underbrace{\langle \cdots \langle}_{n} \{\texttt{val} = e\} \underbrace{\rangle \cdots \rangle}_{n}\}}$$

$$\boxed{\Gamma \vdash^n E : \tau \leadsto e}$$

$$\frac{\Gamma \vdash M : \exists \boldsymbol{b} \,.\, \{R\} \leadsto e \qquad R(l_X) = \llparenthesis \tau \rrparenthesis^n \qquad \lfloor \Gamma \rfloor \vdash \tau :: \bullet}{\Gamma \vdash^n M \,.\, X : \tau \leadsto (\underbrace{\sim \cdots \sim}_{n} (\textbf{unpack}\ (\boldsymbol{b}, y) = e\ \textbf{in}\ y\#l_X))\#\texttt{val}}$$

Essentially, we do something like the following internally:

$$\textbf{val}^n\ X = E \Longrightarrow \textbf{let}\ X = \underbrace{\langle \cdots \langle}_{n}\ E\ \underbrace{\rangle \cdots \rangle}_{n}$$

$$M \,.\, X \Longrightarrow \underbrace{\sim \cdots \sim}_{n}(M \,.\, X)$$

Elaboration Preserves Typing

Theorem

- If $\Gamma \vdash^n E : \tau \rightsquigarrow e$, then $\lfloor\Gamma\rfloor \vdash^n e : \tau$.
- If $\Gamma \vdash M : \xi \rightsquigarrow e$, then $\lfloor\Gamma\rfloor \vdash^0 e : \lfloor\xi\rfloor$.

- $\lfloor\Gamma\rfloor$: Embedding of type env. to System F$\omega^{\langle\rangle}$ ones
- $\lfloor\xi\rfloor$, $\lfloor\Sigma\rfloor$: Embedding of semantic sig. to System F$\omega^{\langle\rangle}$ types

Target Type Safety

Theorem (Preservation of System F$\omega^{\langle\rangle}$).
If $\gamma \vdash^n e : \tau$ and $e \xrightarrow{n} e'$, then $\gamma \vdash^n e' : \tau$.

Theorem (Progress of System F$\omega^{\langle\rangle}$).
If $\vdash^{\geq 1} \gamma$ and $\gamma \vdash^n e : \tau$, then e is a value at stage n, or there exists e' such that $e \xrightarrow{n} e'$.

$\vdash^{\geq 1} \gamma$:$\Leftrightarrow$ all entries of the form $x : \tau^n$ in γ satisfy $n \geq 1$

- Since System F$\omega^{\langle\rangle}$ has type equivalence, proving Inversion Lemma etc. is not so trivial
  - Chapter 30 in TaPL [Pierce 2002] handles this topic

Appendix C:
Cross-Stage Persistence

An Example for CSP: `MakeMap` (Recall)

- Implementing the macro `generate` requires the comparison function on keys (as well as `find_opt` etc.)

```
module MakeMap :> (Key : Ord) -> sig
  type t :: * -> *
  val empty : ∀α. t α
  val find_opt : ∀α. Key.t -> t α -> option α
  …
  ~val generate : ∀α. list (Key.t × α) -> ⟨t α⟩
end = fun(Key : Ord) -> struct
  type t α = Leaf | Node of … (* Balanced binary tree *)
  val empty = Leaf
  val find_opt key map = …
  …
  ~val generate kvs = …
end
```

Comparison function `Key.compare : t -> t -> int` should also be usable at stage 0 here! (not only at stage 1 in ordinary functions)

How to Type-check CSP Items

- We must assert that bodies E of $\mathbf{val}^{\geq n}\ X = E$ depend only on CSP values (i.e. those bound by $\mathbf{val}^{\geq k}$, not by $\mathbf{val}^{k}$)
- Local variables in E of $\mathbf{val}^{\geq n}\ X = E$ should also be allowed

- Extend both source and target with stage var. σ

non-CSP

Can be instantiated to any stage $n'\ (\geq n)$

$$s \ ::= \ n \mid n \dot{+} \sigma \qquad \Gamma \ ::= \cdots \mid \Gamma, \sigma \qquad \Gamma \vdash^{s} E : \tau \rightsquigarrow e$$

$$\gamma \ ::= \cdots \mid \gamma, \sigma \qquad \gamma \vdash^{s} e : \tau$$

How to Extend Target Language with CSP

- Extend System $F\omega^{\langle\rangle}$ terms & types for σ :

$$
\begin{aligned}
e \; ::= \; & \cdots \\
& \mid \langle e \rangle^{\sigma} \mid \; \sim^{\sigma} e && \text{staging constructs with } \sigma \\
& \mid \Lambda \sigma . \, e \mid e \uparrow s && \text{stage variable abs./app.}
\end{aligned}
$$

$$\tau \; ::= \cdots \mid \langle \tau \rangle^{\sigma} \mid \forall \sigma . \, \tau \qquad\qquad \gamma \; ::= \cdots \mid \gamma, \sigma$$

- Extend typing rules:

$$\frac{\sigma \in \gamma \qquad \gamma \vdash^{n+\sigma} e : \tau}{\gamma \vdash^{n} \langle e \rangle^{\sigma} : \langle \tau \rangle^{\sigma}} \qquad\qquad \frac{\sigma \in \gamma \qquad \gamma \vdash^{n} e : \langle \tau \rangle^{\sigma}}{\gamma \vdash^{n+\sigma} \sim^{\sigma} e : \tau}$$

$$\frac{\sigma \notin \gamma \qquad \gamma, \sigma \vdash^{0} e : \tau}{\gamma \vdash^{0} \Lambda \sigma . \, e : \forall \sigma . \, \tau} \qquad\qquad \frac{\gamma \vdash^{0} e : \forall \sigma . \, \tau \qquad \gamma \vdash s}{\gamma \vdash^{0} e \uparrow s : [s/\sigma] e}$$

How to Extend Elaboration for CSP

$$\Sigma \;::=\; \cdots \mid (\!|\, \tau \,|\!)^{\geq n}$$

$$\boxed{\Gamma \vdash B : \exists \boldsymbol{b} \,.\, R \rightsquigarrow e}$$

$$\frac{\sigma \notin \Gamma \qquad \Gamma, \sigma \vdash^{n \dotplus \sigma} E : \tau \rightsquigarrow e}{\Gamma \vdash \mathbf{val}^{\geq n}\ X = E : \exists \epsilon \,.\, \{l_X \mapsto (\!|\, \tau \,|\!)^{\geq n}\} \rightsquigarrow \{l_X \mapsto \Lambda \sigma .\ \langle \underbrace{\langle \cdots \langle}_{n} \{\mathtt{val} = e\} \underbrace{\rangle \cdots \rangle}_{n} \rangle^{\sigma}\}}$$

CSP Does Not Break Type Safety

Theorem
- If $\Gamma \vdash^s E : \tau \rightsquigarrow e$, then $\lfloor \Gamma \rfloor \vdash^s e : \tau$.
- If $\Gamma \vdash M : \xi \rightsquigarrow e$, then $\lfloor \Gamma \rfloor \vdash^0 e : \lfloor \xi \rfloor$.

Theorem (Preservation of System Fω$^{\langle\rangle}$).
If $\gamma \vdash^n e : \tau$ and $e \xrightarrow{n} e'$, then $\gamma \vdash^n e' : \tau$.

Theorem (Progress of System Fω$^{\langle\rangle}$).
If $\vdash^{\geq 1} \gamma$ and $\gamma \vdash^n e : \tau$, then e is a value at stage n, or there exists e' such that $e \xrightarrow{n} e'$.

$\vdash^{\geq 1} \gamma$:⇔ all entries of the form $x : \tau^s$ in γ satisfy $s \geq 1$