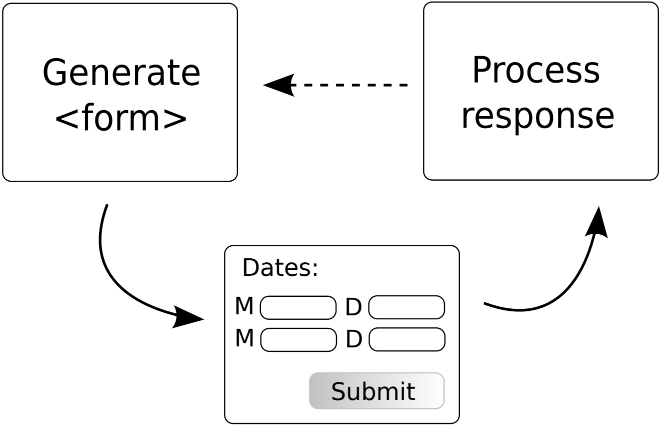
The ML module system

Abstractions for programming in the large

Jeremy Yallop

28 November 2013

Programming with forms



Formlets

```
let date_formlet : date formlet =
  formlet
    <div>
        Month: {input_int ⇒ month}
        Day: {input_int ⇒ day}
    </div>
  yields
    (make_date month day)
```

Formlets

```
let date_formlet : date formlet =
  formlet
    <div>
        Month: {input_int ⇒ month}
        Day: {input_int ⇒ day}
    </div>
  yields
    (make_date month day)
```

```
let date_formlet : date formlet =
  pure (fun ((), month, (), day, ()) → make_date month day)
  ⊗ (tag "div" []
       (pure (fun () month () day () → ((), month, (), day, ()))
        ⊗ text "Month: " ⊗ input_int
        ⊗ text "Day: " ⊗ input_int ⊗ text "\n "))
```

Formlets

```
let travel_formlet : (string * date * date) formlet =
  formlet
    <#> Name: {input ⇒ name}
      <div>
        Arrive: {date_formlet ⇒ arrive}
        Depart: {date_formlet ⇒ depart}
      </div>
      {submit "Submit"}
    </#>
  yields (name, arrive, depart)
```

Formlets

```
let travel_formlet : (string * date * date) formlet =
    formlet
      <#> Name: {input ⇒ name}
          <div>
            Arrive: {date_formlet ⇒ arrive}
            Depart: {date_formlet ⇒ depart}
          </div>
          {submit "Submit"}
      </#>
    yields (name, arrive, depart)
```

```
let travel_formlet : (string × date × date) formlet =
  pure (fun ((), name, (), ((), arrive, (), depart, ()), (), (), ()) →
                                              (name, arrive, depart))
  ⊗ (pure (fun () name () ((), arrive, (), depart, ()) () () () →
                 ((), name, (), ((), arrive, (), depart, ()), (), (), ()))
      ⊗ text "\n   Name: " ⊗ input ⊗ text "\n "
      ⊗ (tag "div" []
           (pure (fun () arrive () depart () → ((), arrive, (), depart, ()))
            ⊗ text "\n     Arrive: " ⊗ date_formlet
            ⊗ text "\n     Depart: " ⊗ date_formlet ⊗ text "\n "))
      ⊗ text "\n   " ⊗ xml (submit "Submit") ⊗ text "\n ")
```

Desugaring formlets

$$
\begin{aligned}
[\![r]\!] &= r^* \\
[\![\textsf{formlet}\ q\ \textsf{yields}\ e]\!] &= \mathit{pure}\ (\textsf{fun}\ q^\dagger \to [\![e]\!])\ \ \otimes\ \ q^\circ \\
\\
s^* &= \mathit{xml\_text}\ s \\
\texttt{\{}e\texttt{\}}^* &= [\![e]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}m_1\ \ldots\ m_k\texttt{</}t\texttt{>})^* &= \mathit{xml\_tag}\ t\ \mathit{ats}\ (\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* \\
(\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* &= m_1^*\ \texttt{@} \cdots \texttt{@}\ m_k^* \\
\\
s^\circ &= \mathit{text}\ s \\
\texttt{\{}e\texttt{\}}^\circ &= \mathit{xml}\ [\![e]\!] \\
\texttt{\{}f \Rightarrow p\texttt{\}}^\circ &= [\![f]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\circ &= \mathit{tag}\ t\ \mathit{ats}\ (\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ &= \mathit{pure}\ (\textsf{fun}\ n_1^\dagger\ \ldots\ n_k^\dagger \to (n_1^\dagger,\ \ldots\ , n_k^\dagger)) \\
&\qquad\quad \otimes\ n_1^\circ\ \ \cdots\ \ \otimes\ n_k^\circ \\
\\
s^\dagger &= \texttt{()} \\
\texttt{\{}e\texttt{\}}^\dagger &= \texttt{()} \\
\texttt{\{}f \Rightarrow p\texttt{\}}^\dagger &= p \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\dagger &= (n_1^\dagger,\ \ldots,\ n_k^\dagger) \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\dagger &= (n_1^\dagger,\ \ldots,\ n_k^\dagger)
\end{aligned}
$$

Desugaring formlets

$$
\begin{aligned}
[\![r]\!] &= r^* \\
[\![\textsf{formlet } q \textsf{ yields } e]\!] &= \mathit{pure}\ (\textsf{fun } q^\dagger \to [\![e]\!])\ \otimes\ q^\circ \\
\\
s^* &= \mathit{xml\_text}\ s \\
\texttt{\{}e\texttt{\}}^* &= [\![e]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}m_1\ \ldots\ m_k\texttt{</}t\texttt{>})^* &= \mathit{xml\_tag}\ t\ \mathit{ats}\ (\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* \\
(\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* &= m_1^*\ \texttt{@} \cdots \texttt{@}\ m_k^* \\
\\
s^\circ &= \mathit{text}\ s \\
\texttt{\{}e\texttt{\}}^\circ &= \mathit{xml}\ [\![e]\!] \\
\texttt{\{}f \Rightarrow p\texttt{\}}^\circ &= [\![f]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\circ &= \mathit{tag}\ t\ \mathit{ats}\ (\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ &= \mathit{pure}\ (\textsf{fun } n_1^\dagger\ \ldots\ n_k^\dagger \to (n_1^\dagger, \ldots, n_k^\dagger)) \\
&\qquad\quad \otimes n_1^\circ\ \cdots\ \otimes n_k^\circ \\
\\
s^\dagger &= \texttt{()} \\
\texttt{\{}e\texttt{\}}^\dagger &= \texttt{()} \\
\texttt{\{}f \Rightarrow p\texttt{\}}^\dagger &= p \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\dagger &= (n_1^\dagger, \ldots, n_k^\dagger) \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\dagger &= (n_1^\dagger, \ldots, n_k^\dagger)
\end{aligned}
$$

Optimising formlets

The `input_int` formlet:

```
let input_int : int formlet =
  formlet <#>{input ⇒ i}</#>
   yields (int_of_string i)
```

Optimising formlets

The `input_int` formlet:

$$\begin{array}{l}\textsf{let}\ \mathrm{input\_int} : \mathrm{int}\ \textsf{formlet} = \\ \quad \textsf{formlet}\ \text{<\#>}\{\mathrm{input} \Rightarrow \mathrm{i}\}\text{</\#>} \\ \quad\ \textsf{yields}\ (\mathrm{int\_of\_string}\ \mathrm{i})\end{array}$$

`input_int` desugared:

$$\mathrm{pure}\ (\textsf{fun}\ \mathrm{i} \to \mathrm{int\_of\_string}\ \mathrm{i}) \otimes (\mathrm{pure}\ (\textsf{fun}\ \mathrm{i} \to \mathrm{i}) \otimes \mathrm{input})$$

Optimising formlets

The `input_int` formlet:

```
let input_int : int formlet =
  formlet <#>{input ⇒ i}</#>
   yields (int_of_string i)
```

`input_int` desugared:

```
pure (fun i → int_of_string i) ⊗ (pure (fun i → i) ⊗ input)
```

`input_int` simplified:

```
pure int_of_string ⊗ input
```

Applicative functor laws

Laws:

$$
\begin{array}{rcll}
\mathit{pure}\ f \otimes \mathit{pure}\ x & \equiv & \mathit{pure}\ (f\ x) & \text{homomorphism} \\
\mathit{pure}\ \mathit{id} \otimes u & \equiv & u & \text{identity} \\
\mathit{pure}\ (\circ) \otimes u \otimes v \otimes w & \equiv & u \otimes (v \otimes w) & \text{composition} \\
u \otimes \mathit{pure}\ x & \equiv & \mathit{pure}\ ((|>)\ x) \otimes u & \text{interchange}
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{id} & = & \mathsf{fun}\ x \rightarrow x \\
(\circ) & = & \mathsf{fun}\ f\ g\ x \rightarrow f\ (g\ x) \\
(|>) & = & \mathsf{fun}\ x\ f \rightarrow f\ x
\end{array}
$$

Normal form:

$$\mathit{pure}\ f \otimes u_1 \otimes u_2 \otimes \ldots \otimes u_n$$

Plan

Aim: justify post-desugaring simplification by showing that the idiom laws hold for formlets

Approach: use the module system to construct the formlet definition from (trivial) components

Benefits: Easy to reason about; easy to substitute components later on without throwing everything away

Module primer (I)

Module definitions

module M : *sig* = *mexp*

Module expressions

struct *decls* end

$\mathtt{M}_1\mathtt{.M}_2\mathtt{.}\ \ldots\ \mathtt{.M}_n$

Using modules

M.x   M.t   M.N

include *mexp*

Signature definitions

module type S = *sigexp*

Signature expressions

sig *decls* end

$\mathtt{M}_1\mathtt{.M}_2\mathtt{.}\ \ldots\ \mathtt{.M}_n\mathtt{.S}$

Using signatures

include *sexp*

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
   | [] → None
   | x :: xs when v = x → Some x
   | x :: xs → assoc v xs
end
```

Module primer (I)

Module definitions

```
module M : sig = mexp
```

Module expressions

```
struct decls end
M1.M2. ... .Mn
```

Using modules

```
M.x    M.t    M.N
include mexp
```

Signature definitions

```
module type S = sigexp
```

Signature expressions

```
sig decls end
M1.M2. ... .Mn.S
```

Using signatures

```
include sexp
```

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
    | [] → None
    | x :: xs when v = x → Some x
    | x :: xs → assoc v xs
end
```

Module primer (I)

Module definitions

module M : *sig* = *mexp*

Module expressions

struct *decls* end

$\texttt{M}_1.\texttt{M}_2.\ \ldots\ .\texttt{M}_n$

Using modules

M.x   M.t   M.N

include *mexp*

Signature definitions

module type S = *sigexp*

Signature expressions

sig *decls* end

$\texttt{M}_1.\texttt{M}_2.\ \ldots\ .\texttt{M}_n.\texttt{S}$

Using signatures

include *sexp*

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
   | [] → None
   | x :: xs when v = x → Some x
   | x :: xs → assoc v xs
end
```

Module primer (I)

Module definitions

`module M : sig = mexp`

Module expressions

`struct decls end`

$\mathtt{M}_1\mathtt{.M}_2\mathtt{.}\ \ldots\ \mathtt{.M}_n$

Using modules

`M.x    M.t    M.N`

`include mexp`

Signature definitions

`module type S = sigexp`

Signature expressions

`sig decls end`

$\mathtt{M}_1\mathtt{.M}_2\mathtt{.}\ \ldots\ \mathtt{.M}_n\mathtt{.S}$

Using signatures

`include sexp`

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
    | [] → None
    | x :: xs when v = x → Some x
    | x :: xs → assoc v xs
end
```

Module primer (I)

Module definitions

module M : *sig* = *mexp*

Module expressions

struct *decls* end

$M_1.M_2.\ \ldots\ .M_n$

Using modules

M.x M.t M.N

include *mexp*

Signature definitions

module type S = *sigexp*

Signature expressions

sig *decls* end

$M_1.M_2.\ \ldots\ .M_n.S$

Using signatures

include *sexp*

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
   | [] → None
   | x :: xs when v = x → Some x
   | x :: xs → assoc v xs
end
```

Module primer (I)

Module definitions

module M : *sig* = *mexp*

Module expressions

struct *decls* end

$M_1.M_2.\ \ldots\ .M_n$

Using modules

M.x M.t M.N

include *mexp*

Signature definitions

module type S = *sigexp*

Signature expressions

sig *decls* end

$M_1.M_2.\ \ldots\ .M_n.S$

Using signatures

include *sexp*

```
module type ASSOC_LIST =
sig
 type α t
 type key
 include Std.MAPPABLE
 val length : t → int
 val assoc : key → α list → α option
end

module Assoc_list : ASSOC_LIST =
struct
 type key = String.t
 type α t = (key × α) list
 include Std.List
 let rec assoc v l = match l with
    | [] → None
    | x :: xs when v = x → Some x
    | x :: xs → assoc v xs
end
```

The idiom interface

```
module type IDIOM =
sig
  type α t
  val pure : α → α t
  val (⊗) : (α → β) t → α t → β t
end
```

Formlets, monolithically

```
module type FORMLET =
sig
  include IDIOM
  val xml : xml → unit t
  val text : string → unit t
  val tag  : tag → attrs → α t → α t
  val input : string t
  val run  : α t → xml × (env → α)
end
```

```
module Formlet : FORMLET =
struct
 type α t =
   int → (xml × (env → α) × int)

 let pure x i = ([], const x, i)

 let (⊗) f p i =
  let (x₁, g, i) = f i in
  let (x₂, q, i) = p i in
   (x₁ @ x₂, (fun env → g env (q env)), i)

  ...

 let run c = let (x, f, _) = c 0 in (x, f)
end
```

The environment idiom

```
module type ENVIRONMENT =
sig
  include IDIOM
  type env = (string × string) list
  val lookup : string → string t
  val run : α t → env → α
end
```

```
module Environment : ENVIRONMENT =
struct
  type env = (string × string) list
  type α t = env → α
  let pure v e = v
  let (⊗) f p e = f e (p e)
  let lookup = List.assoc
  let run v = v
end
```

The environment idiom: laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for the environment idiom.

The environment idiom: laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for the environment idiom.

$$\text{pure f} \otimes \text{pure x}$$

The environment idiom: laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for the environment idiom.

$$\text{pure f} \otimes \text{pure x}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ f\ p\ e \rightarrow f\ e\ (p\ e))\ (\mathsf{fun}\ \_ \rightarrow f)\ (\mathsf{fun}\ \_ \rightarrow x)$$

The environment idiom: laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for the environment idiom.

$$\text{pure f} \otimes \text{pure x}$$

$$\equiv \quad \textit{(definition of } \otimes \textit{ and pure)}$$

$$(\mathsf{fun}\ \text{f p e} \rightarrow \text{f e (p e)})\ (\mathsf{fun}\ \_ \rightarrow \text{f})\ (\mathsf{fun}\ \_ \rightarrow \text{x})$$

$$\equiv \quad \textit{(repeated } \beta\textit{-reduction)}$$

$$\mathsf{fun}\ \text{e} \rightarrow \text{f x}$$

The environment idiom: laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for the environment idiom.

$$\text{pure f} \otimes \text{pure x}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ \text{f p e} \to \text{f e (p e)})\ (\mathsf{fun}\ \_ \to \text{f})\ (\mathsf{fun}\ \_ \to \text{x})$$

$\equiv$ *(repeated β-reduction)*

$$\mathsf{fun}\ \text{e} \to \text{f x}$$

$\equiv$ *(definition of pure)*

$$\text{pure (f x)}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \circledast \text{u} \quad \equiv \quad \text{u}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \otimes \text{u} \quad \equiv \quad \text{u}$$

$$\text{pure id} \otimes \text{u}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \otimes \text{u} \quad \equiv \quad \text{u}$$

$$\text{pure id} \otimes \text{u}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ \text{f p e} \rightarrow \text{f e (p e)})\ (\mathsf{fun}\ \_ \rightarrow \text{id})\ \text{u}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \otimes \text{u} \quad \equiv \quad \text{u}$$

$$\text{pure id} \otimes \text{u}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ \text{f p e} \rightarrow \text{f e (p e)})\ (\mathsf{fun}\ \_ \rightarrow \text{id})\ \text{u}$$

$\equiv$ *(repeated β-reduction)*

$$\mathsf{fun}\ \text{e} \rightarrow \text{id (u e)}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \otimes \text{u} \quad \equiv \quad \text{u}$$

$$\text{pure id} \otimes \text{u}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ \text{f p e} \to \text{f e (p e)})\ (\mathsf{fun}\ \_ \to \text{id})\ \text{u}$$

$\equiv$ *(repeated β-reduction)*

$$\mathsf{fun}\ \text{e} \to \text{id (u e)}$$

$\equiv$ *(definition of id)*

$$\mathsf{fun}\ \text{e} \to \text{u e}$$

The environment idiom: laws

Let's try proving the identity law:

$$\text{pure id} \otimes \text{u} \quad \equiv \quad \text{u}$$

$$\text{pure id} \otimes \text{u}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$(\mathsf{fun}\ f\ p\ e \rightarrow f\ e\ (p\ e))\ (\mathsf{fun}\ \_ \rightarrow \text{id})\ u$$

$\equiv$ *(repeated β-reduction)*

$$\mathsf{fun}\ e \rightarrow \text{id}\ (u\ e)$$

$\equiv$ *(definition of id)*

$$\mathsf{fun}\ e \rightarrow u\ e$$

$\equiv$ *(η-contraction)*

$$u$$

The name generation idiom

```
module type NAMER =
sig
  include IDIOM
  val next_name : string t
  val run : α t → α
end
```

```
module Namer : NAMER =
struct
  type α t = int → α × int
  let pure v i = (v, i)
  let (⊗) f p i = let (f′, i) = f i in
                  let (p′, i) = p i in
                    (f′ p′, i)
  let next_name i =
   ("input_"^string_of_int i, i+1)
  let run v = fst (v 0)
end
```

The XML accumulation idiom

```
module XMLWRITER =
sig
  include IDIOM
  val text : string → unit t
  val xml : xml → unit t
  val tag : tag → attrs → α t → α t
  val run : α t → xml × α
end
```

```
module XmlWriter : XMLWRITER =
struct
  type α t = xml × α
  let pure v = ([], v)
  let (⊗) (x, f) (y, p) = (x @ y, f p)
  let text x = (xml_text x, ())
  let xml x = (x, ())
  let tag t a (x,v) = (xml_tag t a x, v)
  let run v = v
end
```

```
type xml = xml_item list
 and tag = string
 and attrs = (string × string) list
 and xml_item

val tag  : tag → attrs → xml → xml
val text : string → xml
```

Module primer (II)

Module definitions

```
module M (X₁ : sig)
         (X₂ : ... sig)
          ...
         (Xₙ : ... sig)
  : sig = mexp
```

Module expressions

```
mexp (mexp)
```

Signature expressions

```
sigexp
 with type t₁ = type-exp₁
  and type t₂ = type-exp₂
  and ...
  and type tₙ = type-expₙ
```

```
module Assoc_list (Key : KEY) :
   ASSOC_LIST
    with type key = Key.t =
struct
 ...
end

module String_assoc_list =
  Assoc_list(Std.String)

module Int_assoc_list =
   Assoc_list(Std.Int)
```

Module primer (II)

Module definitions

```
module M (X₁ : sig)
         (X₂ : ... sig)
          ...
         (Xₙ : ... sig)
   : sig = mexp
```

Module expressions

```
mexp (mexp)
```

Signature expressions

```
sigexp
 with type t₁ = type-exp₁
  and type t₂ = type-exp₂
  and ...
  and type tₙ = type-expₙ
```

```
module Assoc_list (Key : KEY) :
   ASSOC_LIST
    with type key = Key.t =
struct
 ...
end

module String_assoc_list =
  Assoc_list(Std.String)

module Int_assoc_list =
   Assoc_list(Std.Int)
```

Module primer (II)

Module definitions

```
module M (X₁ : sig)
         (X₂ : ... sig)
          ...
         (Xₙ : ... sig)
   : sig = mexp
```

Module expressions

```
mexp (mexp)
```

Signature expressions

```
sigexp
 with type t₁ = type-exp₁
  and type t₂ = type-exp₂
  and ...
  and type tₙ = type-expₙ
```

```
module Assoc_list (Key : KEY) :
   ASSOC_LIST
    with type key = Key.t =
struct
 ...
end

module String_assoc_list =
  Assoc_list(Std.String)

module Int_assoc_list =
   Assoc_list(Std.Int)
```

Module primer (II)

Module definitions

```
module M (X₁ : sig)
         (X₂ : ... sig)
          ...
         (Xₙ : ... sig)
  : sig = mexp
```

Module expressions

```
mexp (mexp)
```

Signature expressions

```
sigexp
 with type t₁ = type-exp₁
  and type t₂ = type-exp₂
  and ...
  and type tₙ = type-expwₙ
```

```
module Assoc_list (Key : KEY) :
   ASSOC_LIST
    with type key = Key.t =
struct
 ...
end

module String_assoc_list =
  Assoc_list(Std.String)

module Int_assoc_list =
   Assoc_list(Std.Int)
```

Idiom composition

```
module Compose (F : IDIOM) (G : IDIOM) :
sig
  include IDIOM with type α t = (α G.t) F.t
  val refine : α F.t → (α G.t) F.t
end
=
struct
  type α t = (α G.t) F.t
  let pure x = F.pure (G.pure x)
  let (⊗) f x = F.pure (⊗_G) ⊗_F f ⊗_F x
  let refine v = (F.pure G.pure) ⊗_F v
end
```

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$\text{pure f} \otimes \text{pure x}$$

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$\begin{array}{ll} & \text{pure f} \otimes \text{pure x} \\ \equiv & \textit{(definition of } \otimes \textit{ and pure)} \end{array}$$

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$\text{pure f} \otimes \text{pure x}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$\text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)}$$

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$\text{pure f} \otimes \text{pure x}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$\text{F.pure} \ (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)}$$

$\equiv$ *(homomorphism law for F (twice))*

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$\text{pure f} \otimes \text{pure x}$$

$\equiv$ *(definition of $\otimes$ and pure)*

$$\text{F.pure} \ (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)}$$

$\equiv$ *(homomorphism law for F (twice))*

$$\text{F.pure (G.pure f} \otimes_G \text{G.pure x)}$$

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$
\begin{array}{ll}
 & \text{pure f} \otimes \text{pure x} \\
\equiv & \textit{(definition of } \otimes \textit{ and pure)} \\
 & \text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv & \textit{(homomorphism law for F (twice))} \\
 & \text{F.pure (G.pure f } \otimes_G \text{ G.pure x)} \\
\equiv & \textit{(homomorphism law for G)} \\
 & \text{F.pure (G.pure (f x))}
\end{array}
$$

Idiom composition laws

We want to show that the homomorphism law

$$\text{pure f} \otimes \text{pure x} \quad \equiv \quad \text{pure (f v)}$$

holds for any composition of idioms.

$$
\begin{array}{ll}
 & \text{pure f} \otimes \text{pure x} \\
\equiv & \textit{(definition of } \otimes \textit{ and pure)} \\
 & \text{F.pure } (\otimes_G) \otimes_F \text{ F.pure (G.pure f)} \otimes_F \text{ F.pure (G.pure x)} \\
\equiv & \textit{(homomorphism law for F (twice))} \\
 & \text{F.pure (G.pure f} \otimes_G \text{ G.pure x)} \\
\equiv & \textit{(homomorphism law for G)} \\
 & \text{F.pure (G.pure (f x))} \\
\equiv & \textit{(definition of pure)} \\
 & \text{pure (f x)}
\end{array}
$$

Formlets, compositionally

```
module type FORMLET =
sig
  include IDIOM
  val xml : xml → unit t
  val text : string → unit t
  val tag  : tag → attrs → α t → α t
  val input : string t
  val run  : α t → xml × (env → α)
end
```

```
module Formlet : FORMLET =
struct
  include Compose (Namer)
            (Compose (XmlWriter)
               (Environment))
  ...
  let run v =
    let xml, collector =
       XmlWriter.run (Namer.run v)
    in
       (xml, Environment.run collector)
end
```

Closing quote

We propose [...] that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. [...] To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

(On the criteria to be used in decomposing systems into modules
D. L. Parnas (1972))