Delimited overloading

Christophe Troestler

Institut de Mathématique
Université de Mons-Hainaut
Mons, Belgium

OCaml users meeting
February 4, 2009
Grenoble

Acknowledgments

This project was started thanks to the support of

who sponsored

-  Dany Maslowski and
-  Julie De Pril

during their OCaml summer projet 2008.

Outline

Outline

Basic idea

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

Operators,... in the expression `e` are overloaded according to the overloadings defined for the module name `M`.

Basic use

```
Float.(1 + x * f 4)
Float.(4 * u**2 / sqrt(abs alpha))
Hashtbl.(h.(key) <- x)
```

- ☞ Literals, functions, and "constant constructions" substitution.
- ☞ Better readability.

```
Big_int.(if x > 0 then 1 + x else 0)
```

- ☞ Can use the usual comparison operators.

```
Int32.(4 + a.(Int.(1 + x)))
```

- ☞ Nested overloadings.

Basic use

```
Float.(1 + x * f 4)
Float.(4 * u**2 / sqrt(abs alpha))
Hashtbl.(h.(key) <- x)
```

- ☞ Literals, functions, and "constant constructions" substitution.
- ☞ Better readability.

```
Big_int.(if x > 0 then 1 + x else 0)
```

- ☞ Can use the usual comparison operators.

```
Int32.(4 + a.(Int.(1 + x)))
```

- ☞ Nested overloadings.

Basic use

```
Float.(1 + x * f 4)
Float.(4 * u**2 / sqrt(abs alpha))
Hashtbl.(h.(key) <- x)
```

- ☞ Literals, functions, and "constant constructions" substitution.
- ☞ Better readability.

```
Big_int.(if x > 0 then 1 + x else 0)
```

- ☞ Can use the usual comparison operators.

```
Int32.(4 + a.(Int.(1 + x)))
```

- ☞ Nested overloadings.

Static checks & simple optimizations

```
Num.("12345678" + x)
```

☞ Compile time check.

If one writes `Num.("a12")`, when compiling, the following error is issued

```
Parse error:  The string "a12" does not represent a
valid Num.
Preprocessing error on file foo.ml
```

```
Float.((x+1)**2)
```

☞ Simple optimization. The whole expression is substituted by (binding introduced only if needed):

```
let tmp = x +. 1.0 in tmp *. tmp
```

Static checks & simple optimizations

```
Num.("12345678" + x)
```

☞ Compile time check.
If one writes `Num.("a12")`, when compiling, the following error is issued

```
Parse error:  The string "a12" does not represent a
valid Num.
Preprocessing error on file foo.ml
```

```
Float.((x+1)**2)
```

☞ Simple optimization. The whole expression is substituted by (binding introduced only if needed):

```
let tmp = x +. 1.0 in tmp *. tmp
```

Complex numbers

```
Complex.(let z = 3 + 2 I in sin(z * z))
```

- ☞ "I" notation.
- ☞ Let binding are allowed.
- ☞ Complex functions like `sin`, `cos`,... are inlined.

For example,

```
Complex.((2 + 3 I) * f x)
```

is turned into

```
let tmp = f x in
{ Complex.re = (tmp.Complex.re *. 2.0) -.
               (tmp.Complex.im *. 3.0);
  Complex.im = (tmp.Complex.re *. 3.0) +.
               (tmp.Complex.im *. 2.0); }
```

Complex numbers

```
Complex.(let z = 3 + 2 I in sin(z * z))
```

- ☞ "I" notation.
- ☞ Let binding are allowed.
- ☞ Complex functions like `sin`, `cos`,... are inlined.

For example,

```
Complex.((2 + 3 I) * f x)
```

is turned into

```
let tmp = f x in
{ Complex.re = (tmp.Complex.re *. 2.0) -.
                    (tmp.Complex.im *. 3.0);
  Complex.im = (tmp.Complex.re *. 3.0) +.
                    (tmp.Complex.im *. 2.0); }
```

Summary

`pa_do.cmo` provides overloadings for

`Int`	`Float`	`Hashtbl`
`Int32`	`Complex`	`String`
`Int64`		
`Nativeint`		

`pa_do_nums.cmo` provides overloadings for

`Num`
`Ratio`
`Big_int`

☞ Requires `nums.cmo` to be loaded by camlp4 for static checks.

Summary

`pa_do.cmo` provides overloadings for

`Int`	`Float`	`Hashtbl`
`Int32`	`Complex`	`String`
`Int64`		
`Nativeint`		

`pa_do_nums.cmo` provides overloadings for

`Num`
`Ratio`
`Big_int`

☞ Requires `nums.cmo` to be loaded by camlp4 for static checks.

Outline

Concrete syntax v.s. API

Concrete syntax	API
In the source file	In a separate file
Must be repeated in each file	Can be bundled with a library
No possibility of overloading general expressions	One can overload general expressions and perform some optimizations

Demo

Concrete syntax v.s. API

Concrete syntax	API
In the source file	In a separate file
Must be repeated in each file	Can be bundled with a library
No possibility of overloading general expressions	One can overload general expressions and perform some optimizations

Demo

"Set of overloadings"

Constructions that can be overloaded:

literals

```
'x
a.(i)
a.(i) <- x
a.[i]
a.[i] <- x
a.{i}
a.{i} <- x
a := x
a <- x
a.p
```

functions & operators overloadings

general substitutions

How to overload one's own module (1/2)

```
module Foo :
sig
  type t
  val of_int :  int -> t
  val compare :  t -> t -> int
  val add :  t -> t -> t
  val mul :  t -> t -> t
end
```

Overloading with the concrete syntax:

- int literals: `OVERLOAD_INT Foo (of_int)`
- comparison: `OVERLOAD_COMPARISON Foo (compare)`
- functions:

  `OVERLOAD Foo ( ( + ) -> add; ( * ) -> mul )`

How to overload one's own module (1/2)

```
module Foo :
sig
  type t
  val of_int :  int -> t
  val compare :  t -> t -> int
  val add :  t -> t -> t
  val mul :  t -> t -> t
end
```

Overloading with the concrete syntax:

- int literals: `OVERLOAD_INT Foo (of_int)`
- comparison: `OVERLOAD_COMPARISON Foo (compare)`
- functions:

```
OVERLOAD Foo ( ( + ) -> add; ( * ) -> mul )
```

How to overload one's own module (1/2)

```
module Foo :
sig
  type t
  val of_int :  int -> t
  val compare :  t -> t -> int
  val add :  t -> t -> t
  val mul :  t -> t -> t
end
```

Overloading with the concrete syntax:

- int literals: `OVERLOAD_INT Foo (of_int)`
- comparison: `OVERLOAD_COMPARISON Foo (compare)`
- functions:

```
OVERLOAD Foo ( ( + ) -> add; ( * ) -> mul )
```

How to overload one's own module (1/2)

```
module Foo :
sig
  type t
  val of_int :  int -> t
  val compare :  t -> t -> int
  val add :  t -> t -> t
  val mul :  t -> t -> t
end
```

Overloading with the concrete syntax:

- int literals: `OVERLOAD_INT Foo (of_int)`
- comparison: `OVERLOAD_COMPARISON Foo (compare)`
- functions:

```
OVERLOAD Foo ( ( + ) -> add; ( * ) -> mul )
```

How to overload one's own module (2/2)

Remarks

- If `Foo` implements all standard functions `add`, `sub`, `mul`, `div` and `neg` (unary negation):
  ☞ `OVERLOAD_ARITHMETIC Foo`.

- If a new module is implemented:

```
module Special_foo :
sig
  include Foo
  val sub : t -> t -> t
end
```

  ☞ `OVERLOAD Special_foo inherit Foo`
  ☞ `OVERLOAD Special_foo ( ( - ) -> sub )`

How to overload one's own module (2/2)

Remarks

- If `Foo` implements all standard functions `add`, `sub`, `mul`, `div` and `neg` (unary negation):
  ☞ `OVERLOAD_ARITHMETIC Foo.`

- If a new module is implemented:

```
module Special_foo :
sig
  include Foo
  val sub :  t -> t -> t
end
```

  ☞ `OVERLOAD Special_foo inherit Foo`
  ☞ `OVERLOAD Special_foo ( ( - ) -> sub )`

Some more examples

Basin of attraction of Newton's method for $z^3 = 1$.

```
Complex.(
  let z = ref z0 in
  for i = Int.(1) to niter do
    z := (2 * !z + 1 / !z**2) / 3
  done;
  if abs(!z - root0) <= r then Some color0
  ... )
```

Let $D = 1.7$. If $p = [p_0; \ldots; p_n]$ represents the polynomial $\sum_{i=0}^{n} p_i z^i$, its norm is (here) defined by $\|p\| := \sum_{i=0}^{n} |p_i| D^i$

```
let domain = Interval.(17 / 10)
let norm p = Interval.(List.fold_right (fun c n ->
                          abs c + domain * n) p 0)
```

Some more examples

Basin of attraction of Newton's method for $z^3 = 1$.

```
Complex.(
  let z = ref z0 in
  for i = Int.(1) to niter do
    z := (2 * !z + 1 / !z**2) / 3
  done;
  if abs(!z - root0) <= r then Some color0
  ... )
```

Let $D = 1.7$. If $p = [p_0; \ldots; p_n]$ represents the polynomial $\sum_{i=0}^{n} p_i z^i$, its norm is (here) defined by $\|p\| := \sum_{i=0}^{n} |p_i| D^i$

```
let domain = Interval.(17 / 10)
let norm p = Interval.(List.fold_right (fun c n ->
                                  abs c + domain * n) p 0)
```

Outline

Priority and associativity of operators

`pa_infix.cmo`

Concrete syntax:

```
INFIX ( %+ ) RIGHTA HIGHER (+)
INFIX ( ^* ) LEVEL (+)
PREFIX ( /+/ )
POSTFIX ( /// ) LEVEL ( ! )
```

API: treat `a = b |> c` as `a = (b |> c)` and replace `x |> f` by `f x`:

```
open Pa_infix
module L = Level
let l = L.binary (L.Higher L.comparison) ~assoc:L.LeftA in
let expr x y _loc = <:expr< $y$ $x$ >> in
infix "|>" ~expr l
```

Outline

Collaboration with Delimited Overloading

```
DEFINE NEWTON(M,x) = M.( (2 * x + 1 / x**2) / 3 )
```

Use it as

```
NEWTON(Float, r)
NEWTON(Complex, z)
```

- ☞ Poor man defunctorizer;
- ☞ Contrarily to functors, requalifies constants.

Better error reporting

```
DEFINE A(x) =
  let s = ref 0.0 in
  for i = 1 to x do
    s := !s + float i
  done;
  s

let () = print_float(A(100))
```

With the standard macros:

```
File "...", line 8, characters 21-27:
This expression has type float but is here used with type
int
```

Better error reporting

```
DEFINE A(x) =
  let s = ref 0.0 in
  for i = 1 to x do
    s := !s + float i
  done;
  s

let () = print_float(A(100))
```

With Delimited Overloading macros:

```
File "...", line 8, characters 21-27:
Expanding of the macro "A" at the previous location yields
the error:
File "...", line 4, characters 11-13:
This expression has type float but is here used with type
int
```

Outline

Optimization for complex operators

```
Complex.((2 + 3 I) * f x)
```

1. Classify subexpressions according to

```
type t =
  | Zero
  | Re of Ast.expr
  | Im of Ast.expr
  | Cplx of Ast.expr * Ast.expr
  | Unknown of Ast.expr
```

2. Specialize complex functions, introducing bindings as needed.

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}\texttt{(e)}$$

where $\sigma_{\texttt{M}}$: *expression* $\rightarrow$ *expression*

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}\texttt{(0)} = \texttt{0}$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}}$: *expression* $\rightarrow$ *expression*

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}}$: *expression* $\rightarrow$ *expression*

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \rightarrow \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \to \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \to \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \to \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \to \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (1/3)

$$\texttt{M.(e)} \mapsto \sigma_{\texttt{M}}(\texttt{e})$$

where $\sigma_{\texttt{M}} : \textit{expression} \rightarrow \textit{expression}$

Problem encountered:

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int}}$; here $\sigma_{\texttt{Int}}(0) = 0$

```
Int32.(a.(0) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
a.(0l) <- Int32.add 7l x
```

Problem!

☞ Protection of already overloaded expressions

Nested overloading (2/3)

Requirements:

- $\sigma_{\texttt{Int}}(0)$ must be a valid expression;
- it must not change the meaning of the program nor its performance;
- locations must not be affected (for correct error reporting).

Solution:

$$\texttt{M.(e)} \mapsto \texttt{p}(\sigma_{\texttt{M}}(\texttt{e}))$$

where p is an undeclared function name!

☞ p is removed by the surrounding overloading $\Rightarrow$ global flag to know whether to insert π.
☞ **external** p : $\alpha \to \alpha$ = "%identity" forbid some optimizations to take place!

Nested overloading (2/3)

Requirements:

- $\sigma_{\texttt{Int}}(0)$ must be a valid expression;
- it must not change the meaning of the program nor its performance;
- locations must not be affected (for correct error reporting).

Solution:

$$\texttt{M.(e)} \mapsto \texttt{p}(\sigma_{\texttt{M}}(\texttt{e}))$$

where p is an undeclared function name!

☞ p is removed by the surrounding overloading $\Rightarrow$ global flag to know whether to insert π.
☞ **external** $\texttt{p} : \alpha \to \alpha =$ `"%identity"` forbid some optimizations to take place!

Nested overloading (2/3)

Requirements:

- $\sigma_{\texttt{Int}}(0)$ must be a valid expression;
- it must not change the meaning of the program nor its performance;
- locations must not be affected (for correct error reporting).

Solution:

$$\texttt{M.(e)} \mapsto \texttt{p}(\sigma_{\texttt{M}}(\texttt{e}))$$

where p is an undeclared function name!

☞ p is removed by the surrounding overloading $\Rightarrow$ global flag to know whether to insert π.
☞ **external** $\texttt{p} : \alpha \to \alpha =$ `"%identity"` forbid some optimizations to take place!

Nested overloading (2/3)

Requirements:

- $\sigma_{\texttt{Int}}(0)$ must be a valid expression;
- it must not change the meaning of the program nor its performance;
- locations must not be affected (for correct error reporting).

Solution:

$$\texttt{M.(e)} \mapsto \texttt{p}(\sigma_{\texttt{M}}(\texttt{e}))$$

where p is an undeclared function name!

☞ p is removed by the surrounding overloading $\Rightarrow$ global flag to know whether to insert π.
☞ **external** p : $\alpha \to \alpha$ = `"%identity"` forbid some optimizations to take place!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(0)) = \texttt{p}(0)$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(\texttt{0})) = \texttt{p(0)}$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(0)) = \texttt{p(0)}$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(\texttt{0})) = \texttt{p(0)}$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(0)) = \texttt{p(0)}$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(\texttt{0})) = \texttt{p(0)}$

```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

```
p(a.(0) <- Int32.add 7l x)
```

OK!

Nested overloading (3/3)

```
Int32.(a.(Int.(0)) <- 7 + x)
```

↓ $\texttt{Int.(0)} = \texttt{p}(\sigma_{\texttt{Int}}(\texttt{0})) = \texttt{p(0)}$

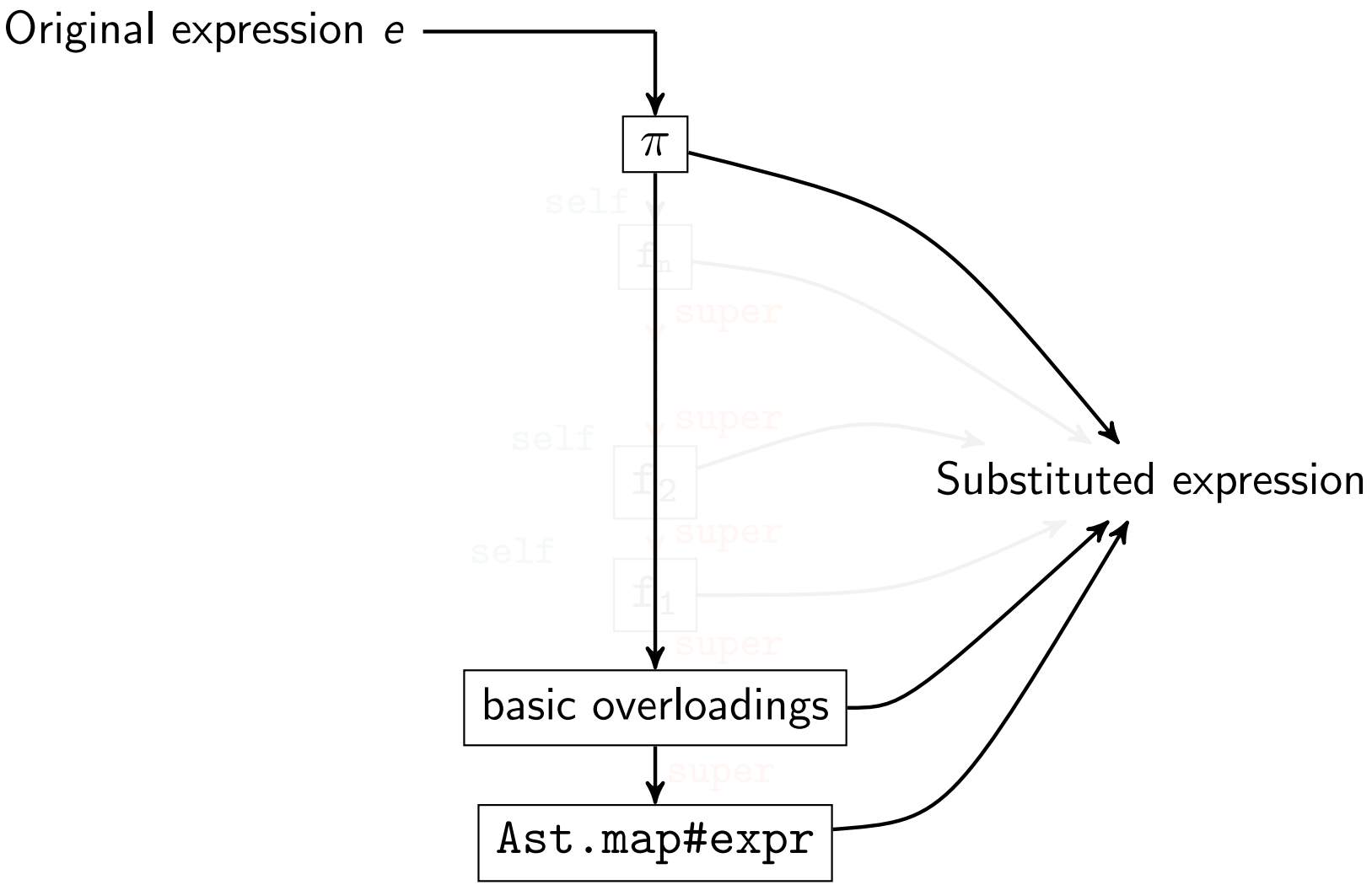
```
Int32.(a.(p(0)) <- 7 + x)
```

↓ apply $\sigma_{\texttt{Int32}}$

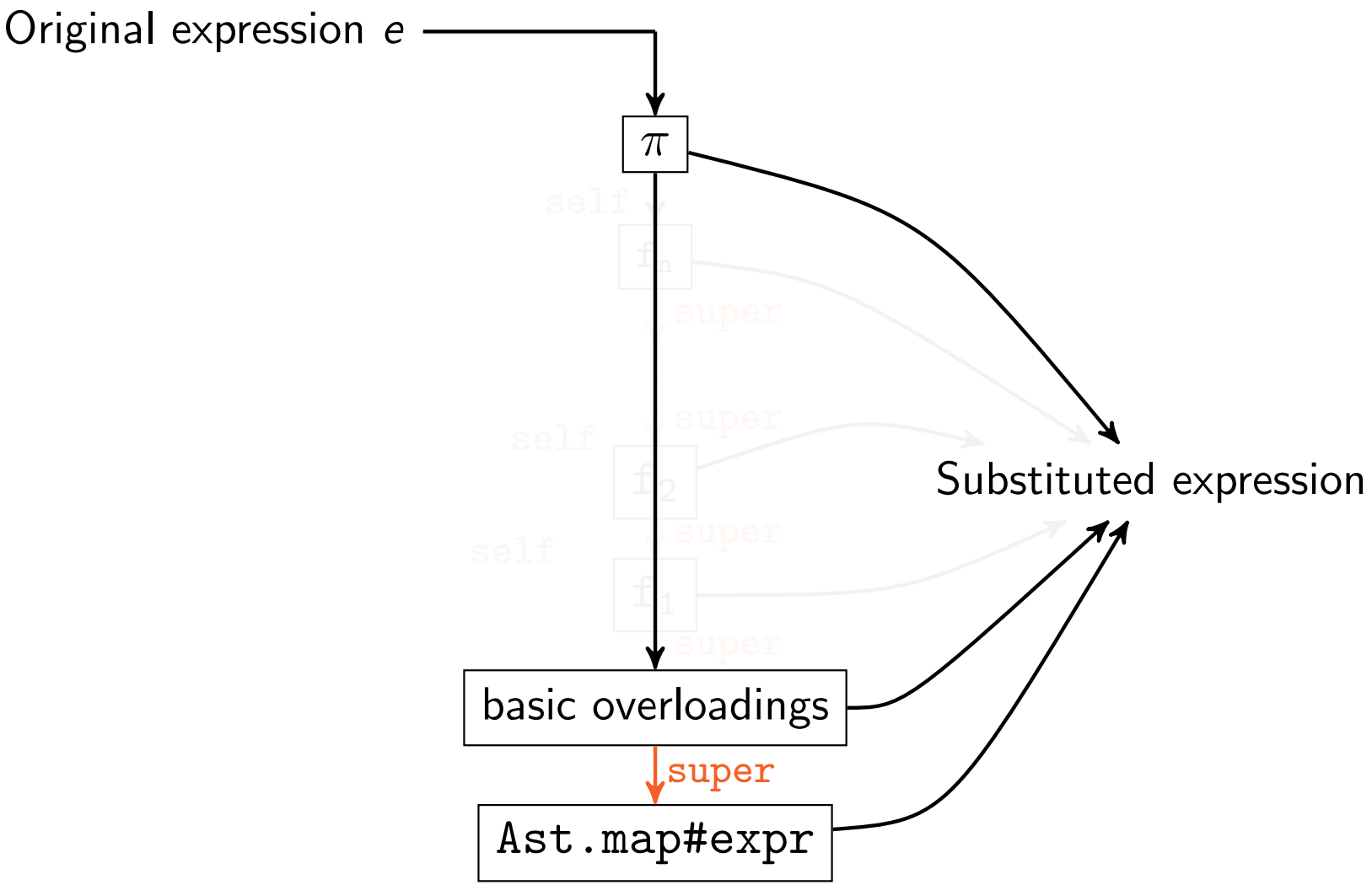
```
p(a.(0) <- Int32.add 7l x)
```

OK!

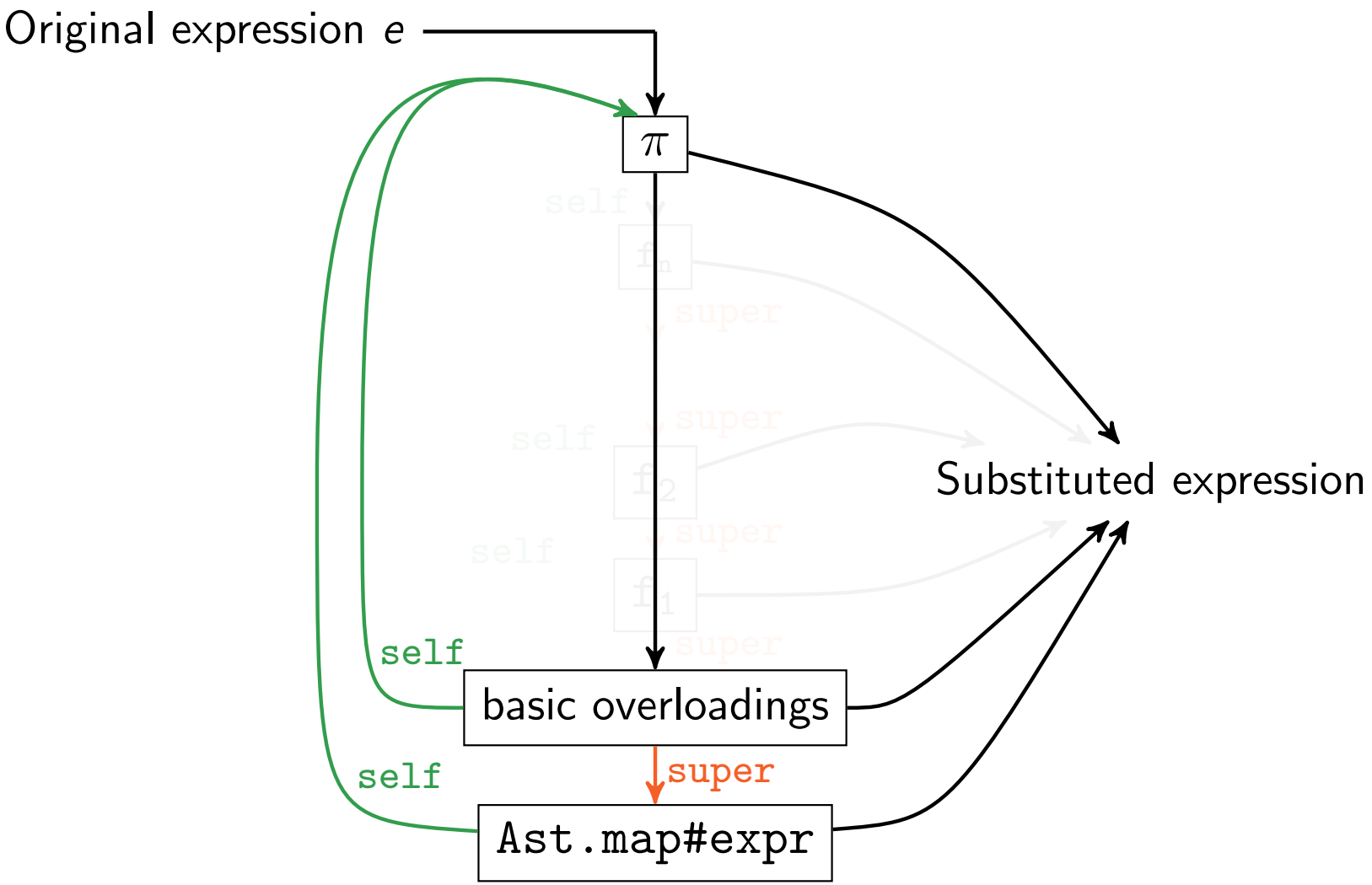
General substitution of expressions `M.(e)`



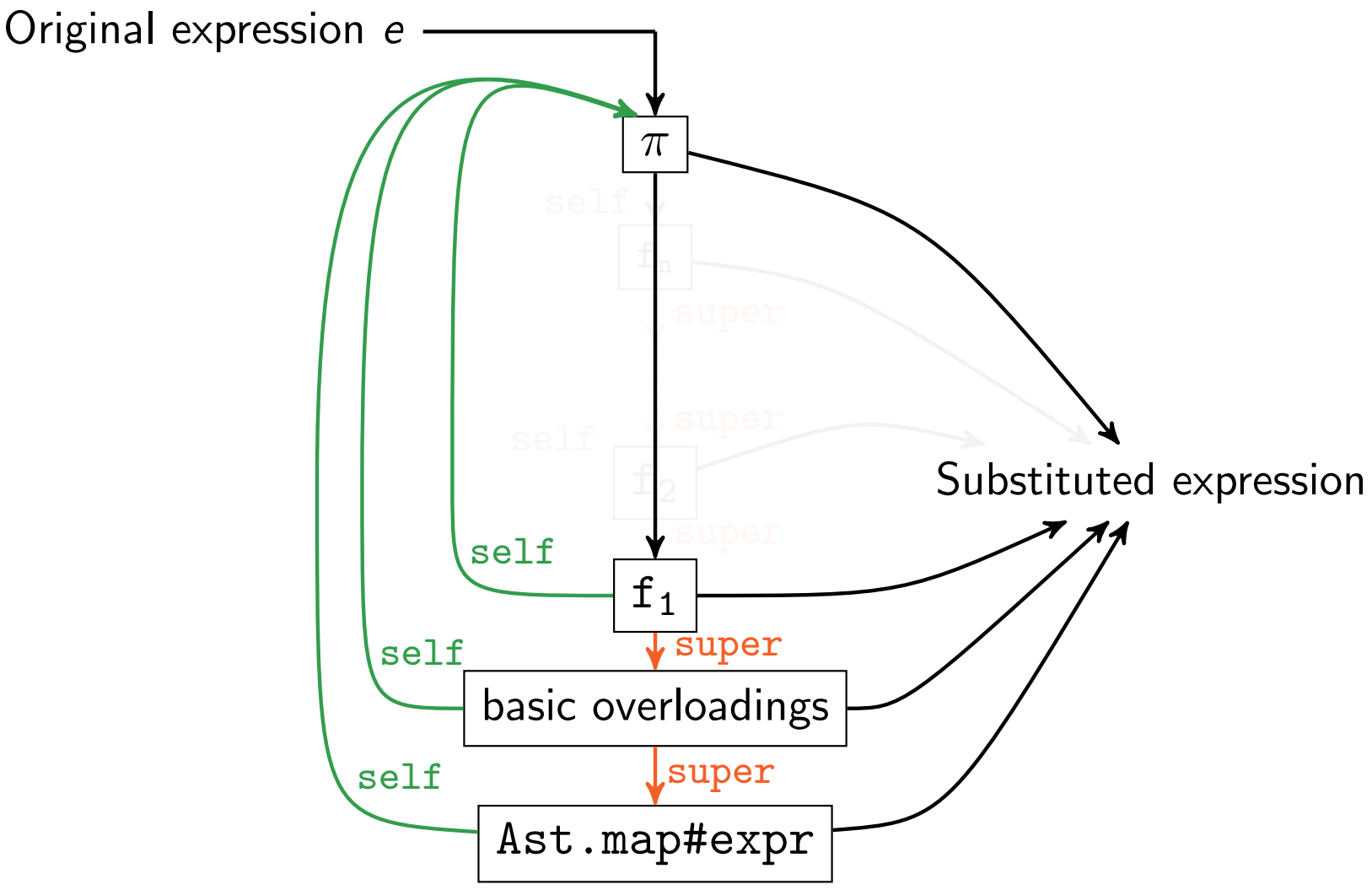
General substitution of expressions `M.(e)`



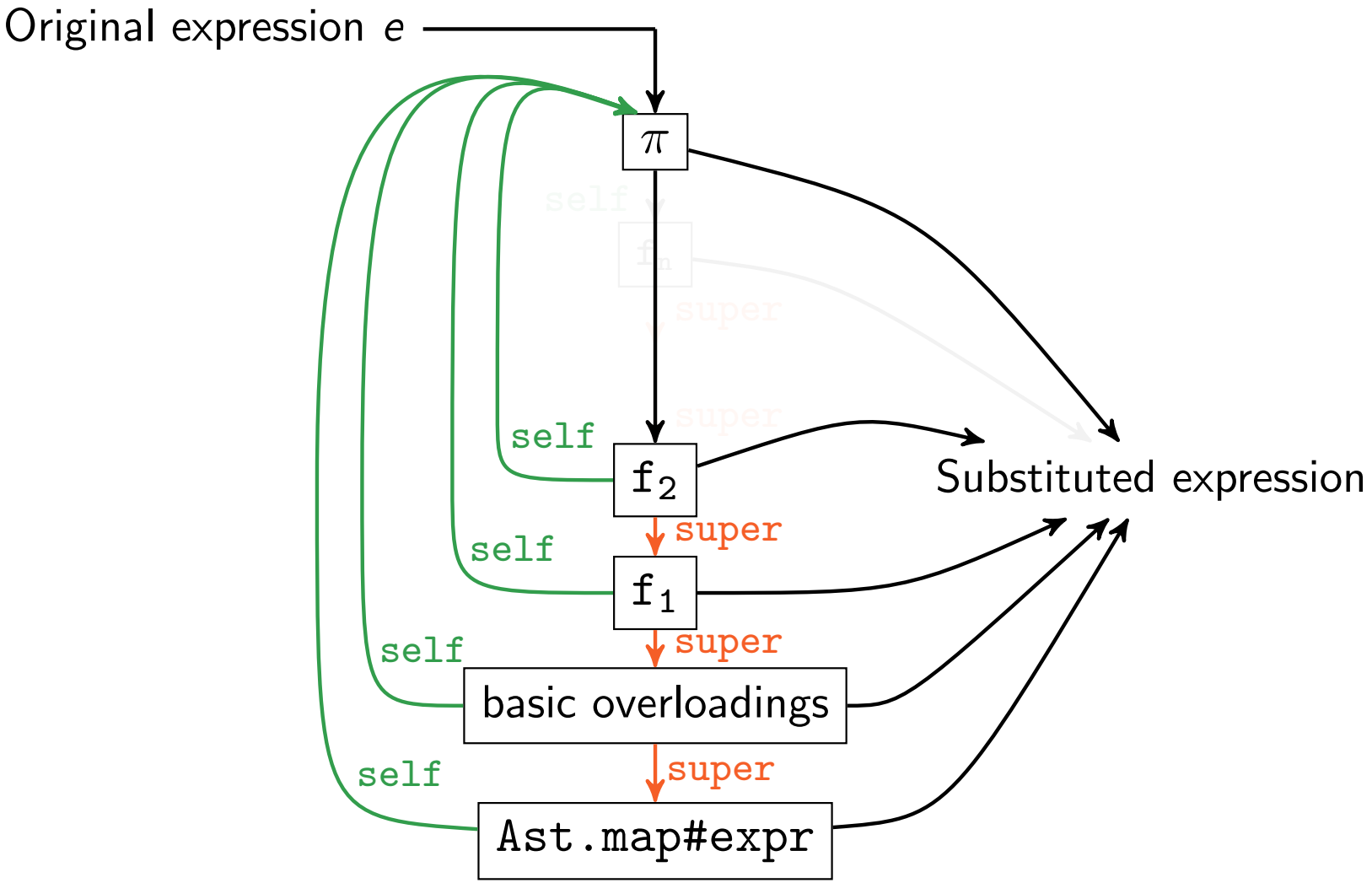
General substitution of expressions `M.(e)`



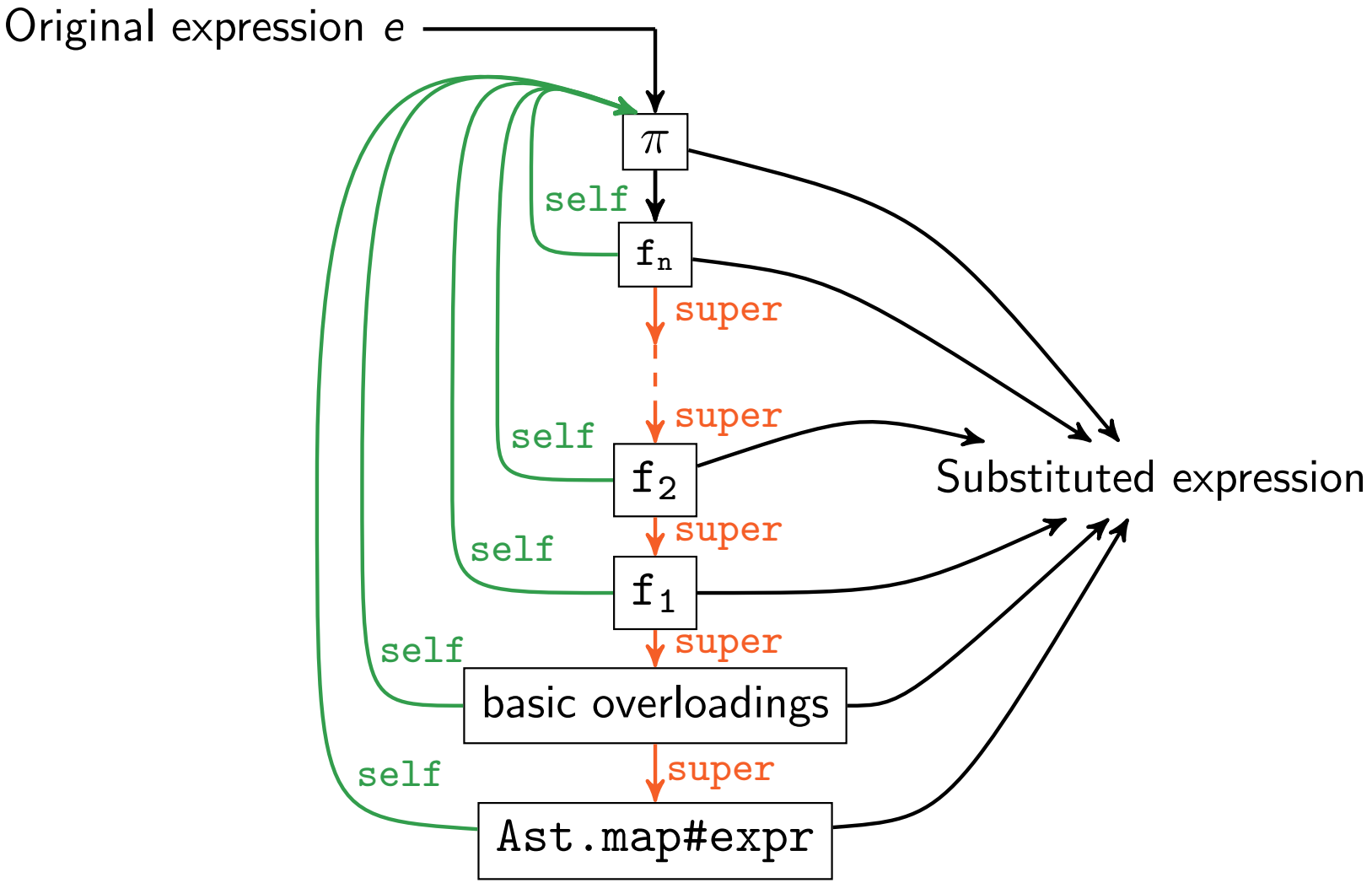
General substitution of expressions `M.(e)`



General substitution of expressions `M.(e)`



General substitution of expressions `M.(e)`



Thank you for your attention...

and to Sylvain Le Gall, Alan Schmitt, and Serge Leblanc for organizing this meeting.