Strongly Typed Nano-passes

Daniel Mercier & Boris Yakobowski

November 7

Context : Infer

- Static analysis tool developed by Facebook
- Tailored for a small amount of false positives
- Facebook's version supports Java, C/C++/Objective-C and Erlang
- Open source: https://github.com/facebook/infer
- Common IR for all languages: SIL
- We added support for Ada using Libadalang

Libadalang

- Parser for the Ada language
- Libadalang’s AST is very close to Ada’s syntax
- Set of semantic queries:
    - Name resolution (with function overloading)
    - Getting the type of an expression
    - Many more

Libadalang to SIL

- SIL is very low level:
    - `Load`
    - `Store`
    - `Call`
- Translation hard to maintain
- Hard to make quick improvements and implement new features

- First step: translate Libadalang to a disambiguated high level IR

Libadalang to AdaIR

- We introduced AdaIR which is a high level intermediate language for Ada
- Tree free of syntactic ambiguities
    - calls / type conversion / dereferences / variable accesses are all explicit
- Two passes:
    - Libadalang to AdaIR
    - AdaIR to SIL

- Still a monolithic style to translate AdaIR to SIL

AdaIR to SIL

- This pass handles too many things at once:
  - Translation of short circuit operators
  - Static evaluation to simplify the AST for the analysis
  - Translation of nested functions to closures
  - Translation of Ada finalization
  - etc

- Unrelated tasks that are performed in one pass

Multiple passes - Unsafe

- A solution would be to use the same AST for all passes

+	−
• We can use the same iterators for all our passes • Don’t need to redefine a new AST	• Don’t known by typing at the start of a pass which nodes are really present in the AST • Need to deal with impossible cases

- But we are interested in having strongly typed passes
- We want to know in each pass which are the possible constructors in the AST

Multiple passes - Safe

- Define a new AST when needed

+	−
• Strongly typed • The type of the AST is exactly the constructors that we have to deal with	• Need to redefine the iterators for each new AST • Cannot reuse helper functions (pretty-printers, evaluators, ...) • Changing one AST type can be a pain to deal with down the line

Nanopass

- Many small localised passes that work on some specific transformations
- Currently 26 passes

```
(env, subps) |> RemoveTasking.apply
|> ExpandBlock.apply |> ExpandWhileLoop.apply
|> ExpandExit.apply
|> ExpandRenames.apply |> ExpandAddressAspect.apply
|> ControlFlowFree.apply |> SideEffectFree.apply
|> NameAsLval.apply |> ExpandSlice.apply
|> LiftStmt.apply |> SimplifyReturn.apply
|> DiscriminantCheck.apply |> InsertDefaultExpr.apply
|> AddChecks.apply |> RemoveCase.apply
|> SimplifyMembership.apply |> RemoveTypePrefix.apply
|> ExpandMembership.apply |> ExpandLoop.apply
|> WrapMod.apply |> StaticEval.apply
|> ArrayAllocLength.apply |> EdgeScope.apply
|> ControlledTypes.apply |> ComputeUplevels.apply
```

Nanopass

- Based on the Nanopass framework for Racket
- Almost each pass refines the type of the AST
- The safe approach but without the downsides

- How to deal with the added boilerplate
    - define a new AST based on the previous one
    - easily write the recursive traverse
- How can we reuse the functionalities written for a previous AST

Normal recursive data type

- Recursive data type

```
type binop = Plus | Minus | Mult
type expr =
  | Var of string
  | Binop of binop * expr * expr
  | IntLit of int
```

- Manual recursion

```
let rec eval = function
  | IntLit i -> Some i
  | Binop (op, l, r) -> (
      match (eval l, eval r) with
      | Some int_l, Some int_r ->
          let op = match op with Plus -> ( + ) | Minus -> ( - ) | Mult -> ( * ) in
          Some (op int_l int_r)
      | _ -> None)
  | _ -> None
```

Recursion schemes

- Factor the recursion out of the type

```
type binop = Plus | Minus | Mult
type expr =
  | Var of string
  | Binop of binop * expr * expr
  | IntLit of int
```

```
type binop = Plus | Minus | Mult
type 'e expr =
  | Var of string
  | Binop of binop * 'e * 'e
  | IntLit of int
```

- Need to define a fixpoint ((([...] expr) expr) expr)

```
(* type fix_expr = ((([...] expr) expr) expr) *)
type fix_expr = fix_expr expr
```

```
    |     type fix_expr = fix_expr expr
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The type abbreviation fix_expr is cyclic
```

Using regular ADTs

- Need to wrap the type around a record

```
type fix_expr = { unfix_expr : fix_expr expr }
```

- Introduces one additional layer at each constructor
- Pattern patching is affected
- Still hard to reuse helper functions (pretty-printers, evaluators, …)

Polymorphic variants

- Use of polymorphic variants

```
type binop = [ `Plus | `Minus | `Mult ]

type 'e expr =
  [ `Var of string
  | `Binop of binop * 'e * 'e
  | `IntLit of int ]

type fix_expr = fix_expr expr
```

- Auto generated map function using ppx_deriving (https://github.com/ocaml-ppx/ppx_deriving)

```
let map_expr poly_e e =
  match e with
  | `Var s -> `Var s
  | `Binop (binop, l, r) ->
      `Binop (binop, poly_e l, poly_e r)
  | `IntLit i -> `IntLit i
```

Recursion schemes - Fold

- Generalization of `List.fold`
  - Takes a value for the `Null` case
  - Takes a function for the `Cons` case
- Instead of using parameters to the function fold, we use the constructors

```
type binop = [ `Plus | `Minus | `Mult ]

type 'e expr =
  [ `Var of string
  | `Binop of binop * 'e * 'e
  | `IntLit of int ]

type fix_expr = fix_expr expr
```

```
let rec fold_expr (f : 'a expr -> 'a) (e : fix_expr) : 'a =
  f (map_expr (fold_expr f) e)

let eval (e : fix_expr) : int option =
  let f (e : int option expr) : int option =
    match e with
    | `IntLit i -> Some i
    | `Binop (`Plus, Some l, Some r) -> Some (l + r)
    | `Binop (`Minus, Some l, Some r) -> Some (l - r)
    | `Binop (`Mult, Some l, Some r) -> Some (l * r)
    | _ -> None
  in
  fold_expr f e
```

Nanopass - Example

- Transform binary operators to N-ary ones

```
type 'e expr_2 =
  [ `Var of string | `Nop of binop * 'e list | `IntLit of int ]

type fix_expr_2 = fix_expr_2 expr_2

let to_expr_2 (e : fix_expr) : fix_expr_2 =
  let f (e : fix_expr_2 expr) : fix_expr_2 =
    match e with
    | `Binop (op1, `Nop (op2, l), `Nop (op3, r)) when op1 = op2 && op2 = op3 ->
        `Nop (op1, l @ r)
    | `Binop (op1, l, `Nop (op2, r)) when op1 = op2 ->
        `Nop (op1, l :: r)
    | `Binop (op1, `Nop (op2, l), r) when op1 = op2 ->
        `Nop (op1, l @ [ r ])
    | `Binop (op, l, r) ->
        `Nop (op, [ l; r ])
    | (`Var _ | `IntLit _) as common ->
        common
  in
  fold_expr f e
```

- Need to redefine all the constructors
- Manual match on common constructors

Nanopass – PPX

- Derive a new AST from the previous one:
    - add new constructors
    - delete existing constructors
    - update existing constructors
- Syntax uses a record with fields called `add`, `del` and `update`

```
type 'e expr = { del : [ `Binop of binop * 'e list ]; add : [ `Nop of binop * 'e list ] } [@@deriving map]

let rec fold_expr f e = f (map_expr (fold_expr f) e)
```

- The ppx also creates a new type called `expr_common`

```
| (`Var _ | `IntLit _) as common -> common
```

```
| #expr_common as common -> common
```

- We can later change a constructor in one AST without impacting too much the following ASTs

Nanopass - Skeleton

- Example of the definition of a pass as we are writing them

```
module ControlledTypes = struct
  module Pre = EdgeScope.L

  module%language L = struct
      (* Cannot write [include Pre] here because of a current limitation in our ppx *)
      include EdgeScope.L

      type 'e t = {del: [...]; add: [...]} [@@deriving map]

      (* For now, need to manually write fold *)
      let fold [...] = [...]
  end

  let insert_finalization e = [...]

  let apply e = Pre.fold insert_finalization e
end
```

Recursion Schemes

- What if we need to match on more than one level of constructors?
- What if the translation we want to perform does not work with a simple fold?

- Two other functions will help us for that:
    - Unfold
    - Refold
- We use `unfold` to translate from top to bottom
- We use `refold` when we need to translate both top down and bottom up

Recursion Schemes - Unfold

- Translation of the Ada Exit statement

```
module%language L1 = struct
  type 's stmt = [ `Exit | `Loop of 's | `Label of Label.t | `Goto of Label.t ]

  and 's stmts = 's stmt list [@@deriving map]

  type fix_stmts = fix_stmts stmts
end

module%language L2 = struct
  include L1

  type 's stmt = { del : [ `Exit ] }

  and 's stmts = 's stmt list [@@deriving map]

  type fix_stmts = fix_stmts stmts

  let rec unfold_stmts (f : 'a -> 'a stmts) (e : 'a) : fix_stmts =
    map_stmts (unfold_stmts f) (f e)
end
```

```
let enrich data x = (data, x)

let expand_exit (stmts : L1.fix_stmts) : L2.fix_stmts =
  let f ((current_loop, stmts) : Label.t option * L1.fix_stmts) :
      (Label.t option * L1.fix_stmts) L2.stmts =
    let aux s =
      match s with
      | `Loop s ->
          let loop_label = Label.mk_fresh () in
          [ `Loop (Some loop_label, s); `Label loop_label ]
      | `Exit ->
          [ `Goto (Option.get current_loop) ]
      | #L2.stmt_common as common ->
          [ L2.map_stmt (enrich current_loop) common ]
      in
      List.concat_map aux stmts
  in
  L2.unfold_stmts f (None, stmts)
```

Nanopass - Refold

- Very useful when a pass generates statements from an expression
- On the way down the function generates statements attached to each expression
- On the way up, another function concatenates the statements attached to the expression
- We will see later that this can also be useful for composing passes

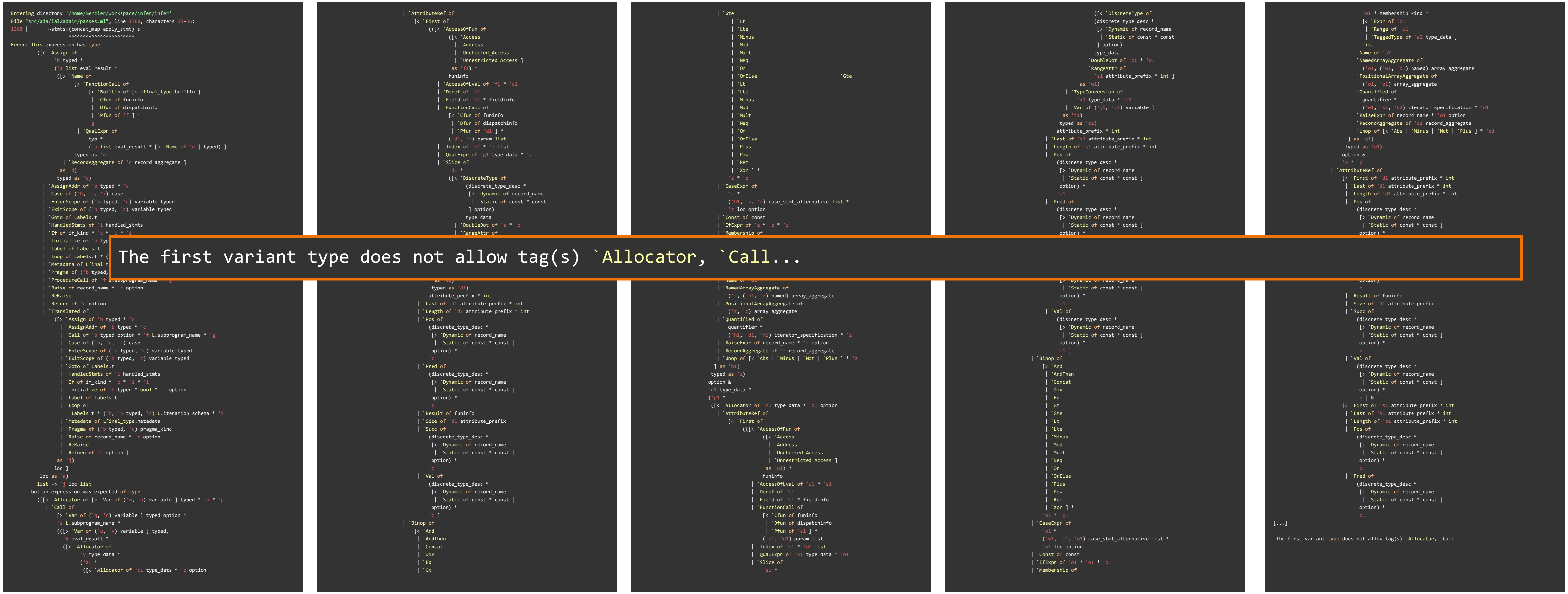
Nanopass - Reusing functions

- We wrote an evaluator for a specific language
- Now we want to reuse the evaluator with a language that has different constructors
- Simply need to write the missing cases

```
module%language L0 = struct
  type binop = [ `Plus | `Minus | `Mult ]
  type 'e expr = [ `Binop of binop * 'e * 'e ]

  let f_eval = function
    | `IntLit i -> Some i
    | `Binop (`Plus, Some l, Some r) -> Some (l + r)
    | `Binop (`Minus, Some l, Some r) -> Some (l - r)
    | `Binop (`Mult, Some l, Some r) -> Some (l * r)
    | _ -> None
end
```

```
module%language L1 = struct
  include L0

  type unop = [ `Plus | `Minus ]
  type 'e expr = { add : [ `Unop of unop * 'e ] } [@@deriving map]

  let rec fold_expr f_e e = f_e (map_expr (fold_expr f_e) e)

  let f_eval = function
    | `Unop (`Plus, Some i) -> Some i
    | `Unop (`Minus, Some i) -> Some (-i)
    | `Unop _ -> None
    | #L0.expr as expr -> L0.f_eval expr

  let eval e : int option = fold_expr f_eval e
end
```

Nanopass - Pitfalls



The first variant type does not allow tag(s) `Allocator, `Call...

Nanopass - Pitfalls

- Trying to understand the typing error can be difficult

```
let to_expr_2 e =
  let f e =
    match e with
    | `Binop (op1, `Binop (op2, l), `Binop (op3, r))
      when op1 = op2 && op2 = op3 ->
        `Nop (op1, l @ r)
    | `Binop (op1, l, `Binop (op2, r)) when op1 = op2 ->
        `Nop (op1, l :: r)
    | `Binop (op1, `Binop (op2, l), r) when op1 = op2 ->
        `Nop (op1, l @ [ r ])
    | `Binop (op, l, r) ->
        `Nop (op, [ l; r ])
    | (`Var _ | `IntLit _) as common ->
        common
  in
  fold_expr f e
```

```
type binop = [ `Plus | `Minus | `Mult ]
type 'e expr = [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
type fix_expr = fix_expr expr

type 'e expr_2 = [ `Var of string | `Nop of binop * 'e list | `IntLit of int ]

type fix_expr_2 = fix_expr_2 expr_2
```

```
fix_expr ->
([> `Binop of binop * 'a list
  | `IntLit of int
  | `Nop of binop * 'a list
  | `Var of string ]
 as 'a)
```

Nanopass - Pitfalls

- Adding type annotations to the pass makes it clear that it is not what we expected

```
let to_expr_2 (e : fix_expr) : fix_expr_2 =
  let f e =
    match e with
    | `Binop (op1, `Binop (op2, l), `Binop (op3, r))
      when op1 = op2 && op2 = op3 ->
        `Nop (op1, l @ r)
    | `Binop (op1, l, `Binop (op2, r)) when op1 = op2 ->
        `Nop (op1, l :: r)
    | `Binop (op1, `Binop (op2, l), r) when op1 = op2 ->
        `Nop (op1, l @ [ r ])
    | `Binop (op, l, r) ->
        `Nop (op, [ l; r ])
    | (`Var _ | `IntLit _) as common ->
        common
  in
  fold_expr f e
```

```
type binop = [ `Plus | `Minus | `Mult ]
type 'e expr = [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
type fix_expr = fix_expr expr

type 'e expr_2 = [ `Var of string | `Nop of binop * 'e list | `IntLit of int ]

type fix_expr_2 = fix_expr_2 expr_2
```

```
|     fold_expr f e
      ^^^^^^^^^^^^^
Error: This expression has type
      [> `Binop of binop * 'a list
       | `IntLit of int
       | `Nop of binop * 'a list
       | `Var of string ] as 'a
      but an expression was expected of type fix_expr_2
      The second variant type does not allow tag(s) `Binop
```

Nanopass - Pitfalls

- Adding the type annotation to the nested function clearly shows where is the error

```
let to_expr_2 (e : fix_expr) : fix_expr_2 =
  let f (e : fix_expr_2 expr) : fix_expr_2 =
    match e with
    | `Binop (op1, `Binop (op2, l), `Binop (op3, r))
      when op1 = op2 && op2 = op3 ->
        `Nop (op1, l @ r)
    | `Binop (op1, l, `Binop (op2, r)) when op1 = op2 ->
        `Nop (op1, l :: r)
    | `Binop (op1, `Binop (op2, l), r) when op1 = op2 ->
        `Nop (op1, l @ [ r ])
    | `Binop (op, l, r) ->
        `Nop (op, [ l; r ])
    | (`Var _ | `IntLit _) as common ->
        common
  in
  fold_expr f e
```

```
type binop = [ `Plus | `Minus | `Mult ]
type 'e expr = [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
type fix_expr = fix_expr expr

type 'e expr_2 = [ `Var of string | `Nop of binop * 'e list | `IntLit of int ]

type fix_expr_2 = fix_expr_2 expr_2
```

```
|     | `Binop (op1, `Binop (op2, l), `Binop (op3, r))
                ^^^^^^^^^^^^^^^
Error: This pattern matches values of type [? `Binop of 'a ]
       but a pattern was expected which matches values of type fix_expr_2
       The second variant type does not allow tag(s) `Binop
```

Nanopass - Performances

- Each pass is traversing the whole tree and is recreating a new tree
- The solution is to compose passes
- Very similar to the deforestation optimization

- Almost all our passes are linear in the size of the tree
- Libadalang is doing the heavy part of the job (name resolution, typing, ...), which is not linear.
- Nanopasses are taking about 10% of the time spent in the translation

Nanopass - Composition

- Combine two passes into one
- One pass translates unary operators to binary ones
- Second pass inlines the static evaluation of the expression

```
type binop = [ `Plus | `Minus | `Mult ]
type unop = [ `Plus | `Minus ]

type 'e expr =
  [ `Var of string
  | `Binop of binop * 'e * 'e
  | `Unop of unop * 'e
  | `IntLit of int ] [@@deriving map]

type fix_expr = fix_expr expr

type 'e expr_2 =
  [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
[@@deriving map]

type fix_expr_2 = fix_expr_2 expr_2
```

```
let translate_unop : fix_expr_2 expr -> fix_expr_2 = function
  | `Unop (`Minus, e) -> `Binop (`Minus, `IntLit 0, e)
  | `Unop (`Plus, e) | ((`Var _ | `IntLit _ | `Binop _) as e) -> e

let static_eval : fix_expr_2 expr_2 -> fix_expr_2 = function
  | `Binop (`Plus, `IntLit l, `IntLit r) -> `IntLit (l + r)
  | `Binop (`Minus, `IntLit l, `IntLit r) -> `IntLit (l - r)
  | `Binop (`Mult, `IntLit l, `IntLit r) -> `IntLit (l * r)
  | (`Binop _ | `Var _ | `IntLit _) as e -> e
```

Nanopass - Composition

- Combine two passes into one
- One pass translates unary operators to binary ones
- Second pass inlines the static evaluation of the expression

```
type binop = [ `Plus | `Minus | `Mult ]
type unop = [ `Plus | `Minus ]

type 'e expr =
  [ `Var of string
  | `Binop of binop * 'e * 'e
  | `Unop of unop * 'e
  | `IntLit of int ] [@@deriving map]

type fix_expr = fix_expr expr

type 'e expr_2 =
  [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
[@@deriving map]

type fix_expr_2 = fix_expr_2 expr_2
```

```
let translate_unop : fix_expr_2 expr -> fix_expr_2 = function
  | `Unop (`Minus, e) -> `Binop (`Minus, `IntLit 0, e)
  | `Unop (`Plus, e) | ((`Var _ | `IntLit _ | `Binop _) as e) -> e

let static_eval : fix_expr_2 expr_2 -> fix_expr_2 = function
  | `Binop (`Plus, `IntLit l, `IntLit r) -> `IntLit (l + r)
  | `Binop (`Minus, `IntLit l, `IntLit r) -> `IntLit (l - r)
  | `Binop (`Mult, `IntLit l, `IntLit r) -> `IntLit (l * r)
  | (`Binop _ | `Var _ | `IntLit _) as e -> e

let not_composed (e : fix_expr) : fix_expr_2 =
  fold_expr translate_unop e |> fold_expr_2 static_eval
```

Nanopass - Composition

- Combine two passes into one
- One pass translates unary operators to binary ones
- Second pass inlines the static evaluation of the expression

```
type binop = [ `Plus | `Minus | `Mult ]
type unop = [ `Plus | `Minus ]

type 'e expr =
  [ `Var of string
  | `Binop of binop * 'e * 'e
  | `Unop of unop * 'e
  | `IntLit of int ] [@@deriving map]

type fix_expr = fix_expr expr

type 'e expr_2 =
  [ `Var of string | `Binop of binop * 'e * 'e | `IntLit of int ]
[@@deriving map]

type fix_expr_2 = fix_expr_2 expr_2
```

```
let translate_unop : fix_expr_2 expr -> fix_expr_2 = function
  | `Unop (`Minus, e) -> `Binop (`Minus, `IntLit 0, e)
  | `Unop (`Plus, e) | ((`Var _ | `IntLit _ | `Binop _) as e) -> e

let static_eval : fix_expr_2 expr_2 -> fix_expr_2 = function
  | `Binop (`Plus, `IntLit l, `IntLit r) -> `IntLit (l + r)
  | `Binop (`Minus, `IntLit l, `IntLit r) -> `IntLit (l - r)
  | `Binop (`Mult, `IntLit l, `IntLit r) -> `IntLit (l * r)
  | (`Binop _ | `Var _ | `IntLit _) as e -> e

let not_composed (e : fix_expr) : fix_expr_2 =
  fold_expr translate_unop e |> fold_expr_2 static_eval

let composed (e : fix_expr) : fix_expr_2 =
  fold_expr (fun x -> static_eval (translate_unop x)) e
```

Nanopass - Composition

- Not always easy to combine passes
- `fold` followed by an `unfold` should be swapped (if possible) to use a `refold`
- A pass with a `fold` can sometimes be written with `unfold` to be composed
- Not possible without changing the pass in some cases
- We did not try to compose too many passes
    - but this is definitely possible

Nanopass - Summary

- We can write passes in isolation without too much boilerplate
- We are able to reuse helper functions written for one language for other languages
- Each pass is strongly typed and impossible cases are avoided by typing
- Recursive traverse is easily redefined for each new AST using recursion schemes
- Not too much PPX magic, we are mostly using existing OCaml features
- The function encoding the pass does not contain recursive calls
    - Makes the code more readable

Future improvements

- Generate the fixed version of the types
- Generate the different iterators
    - Need to write the entry point of the AST to define the higher level iterators
- Generate the types for the passes
    - Not always easy to write as polymorphic variant should not be closed
- Combine passes

References

- Original Nanopass framework: https://nanopass.org/
- Matryoshka (Recursion scheme written in scala): https://github.com/precog/matryoshka
    - https://github.com/precog/matryoshka#external-resources
- Efficient Nanopass compilers in Scala: https://github.com/sellout/recursion-scheme-talk/blob/master/nanopass-compiler-talk.org
- Different approach to the Nanopass framework in OCaml: https://github.com/nanocaml/nanocaml
- Examples for some morphisms in Scala: https://free.cofree.io/2017/11/13/recursion/
- Initial discussions on code reuse: http://www.yakobowski.org/research.html#variants-jfla

Details - Morphisms

- There are multiple possible folds, unfolds and refolds
- Simplest form are
    - catamorphism (what we called fold until now)
    - anamorphism (unfold)
    - hylomorphism (refold)
- But there are many other forms
- For example, to reuse our `eval` function to write `static_eval` we could have used zygomorphism
- One very useful morphism is paramorphism which allows us to match on the whole original tree while translating bottom-up.

Details - Morphisms

Recursion Schemes

folds (tear down a structure) ↔ **unfolds** (build up a structure)

algebra f a → Fix f → a ↔ *coalgebra f a* → a → Fix f

	folds	unfolds	
generalized (f w ⇝ w f) → (f (w α) → β)	**cata**morphism f a → a	**ana**morphism a → f a	**g**eneralized (m f ⇝ f m) → (α → f (m β))
	prepromorphism* … after applying a NatTrans (f a → a) → (f ⇝ f)	**postpro**morphism* … before applying a NatTrans (a → f a) → (f ⇝ f)	
	paramorphism* … with primitive recursion f (Fix f × a) → a	**apo**morphism* … returning a branch or single level a → f (Fix f ∨ a)	
	zygomorphism* … with a helper function (f b → b) → (f (b × a) → a)	**g apo**morphism (b → f b) → (a → f (b ∨ a))	
g histomorphism (f h ⇝ h f) → (f (w a) → a)	**histo**morphism … with prev. answers it has given f (w a) → a	**futu**morphism … multiple levels at a time a → f (m a)	**g futu**morphism (h f ⇝ f h) → (a → f (m a))

refolds (build up then tear down a structure)

algebra g b → (f ⇝ g) → *coalgebra f a* → a → b

hylomorphism cata; ana		**g**eneralized apply the generalizations for both the relevant fold and unfold
dynamorphism histo; ana	**codyna**morphism cata; futu	
chronomorphism histo; futu		
Elgot algebra … may short-circuit while building cata; a → b ∨ f a	**coElgot** algebra … may short-circuit while tearing a × g b → b; ana	

reunfolds (tear down then build up a structure)

coalgebra g b → (a → b) → *algebra f a* → Fix f → Fix g

metamorphism ana; cata	**g**eneralized apply … both … [un]fold

combinations (combine two structures)

algebra f a → Fix f → Fix f → a

zippamorphism f a → a
mergamorphism … which may fail to combine (f (Fix f) × f (Fix f)) ∨ f a → a

others

synchromorphism ???
exomorphism ???
mutumorphism ???

Stolen from Edward Kmett's http://comonad.com/reader/2009/recursion-schemes/

* This gives rise to a family of related recursion schemes, modeled in recursion-schemes with distributive law combinators

These can be combined in various ways. For example, a "zygohistomorphic prepromorphism" combines the zygo, histo, and prepro aspects into a signature like *(f b → b) → (f ⇝ f) → (f (w (b × a)) → a) → Fix f → a*

https://github.com/precog/matryoshka/blob/master/resources/recursion-schemes.pdf

Thank you

Daniel Mercier & Boris Yakobowski

mercier@adacore.com
yakobowski@adacore.com

Nanopass - Pitfalls

- We use a global environment for the translation
  - default initialization for types, body of type predicates, etc
- A pass should also be applied to the syntactic nodes present in the environment
- In the end the environment was translated too many times
  - Same global environment for each compilation unit
  - The passes are applied to the pair `global environment × compilation unit`
- How should passes that use the environment be written?
  - Use `unfold` for those passes

Nanopasses - Mutually recursive types

```
type binop = [ `Plus | `Minus | `Mult ]

type 'e expr_node =
  [ `FunctionCall of string * 'e list
  | `Binop of binop * 'e * 'e
  | `Var of string ]

and ('e, 's) stmt = [ `Call of string * 'e list | `If of 'e * 's * 's ]

and ('e, 's) expr = 's * 'e expr_node

and ('e, 's) stmts = ('e, 's) stmt list [@@deriving map]

type fix_expr = (fix_expr, fix_stmts) expr

and fix_stmts = (fix_expr, fix_stmts) stmts

let fold (f_expr : ('e, 's) expr -> 'e) (f_stmts : ('e, 's) stmts -> 's)
    (s : fix_stmts) : 'b =
  let rec fold_expr e = f_expr (map_expr fold_expr fold_stmts e)
  and fold_stmts s = f_stmts (map_stmts fold_expr fold_stmts s) in
  fold_stmts s
```

- We always use it in practice
- Very similar to the non mutually recursive case
- Not all syntactic categories need to be mutually recursive
- Some passes only visit one syntactic category
- Optimized iterators may stop the recursion early