Compilation

Explicitation des fermetures

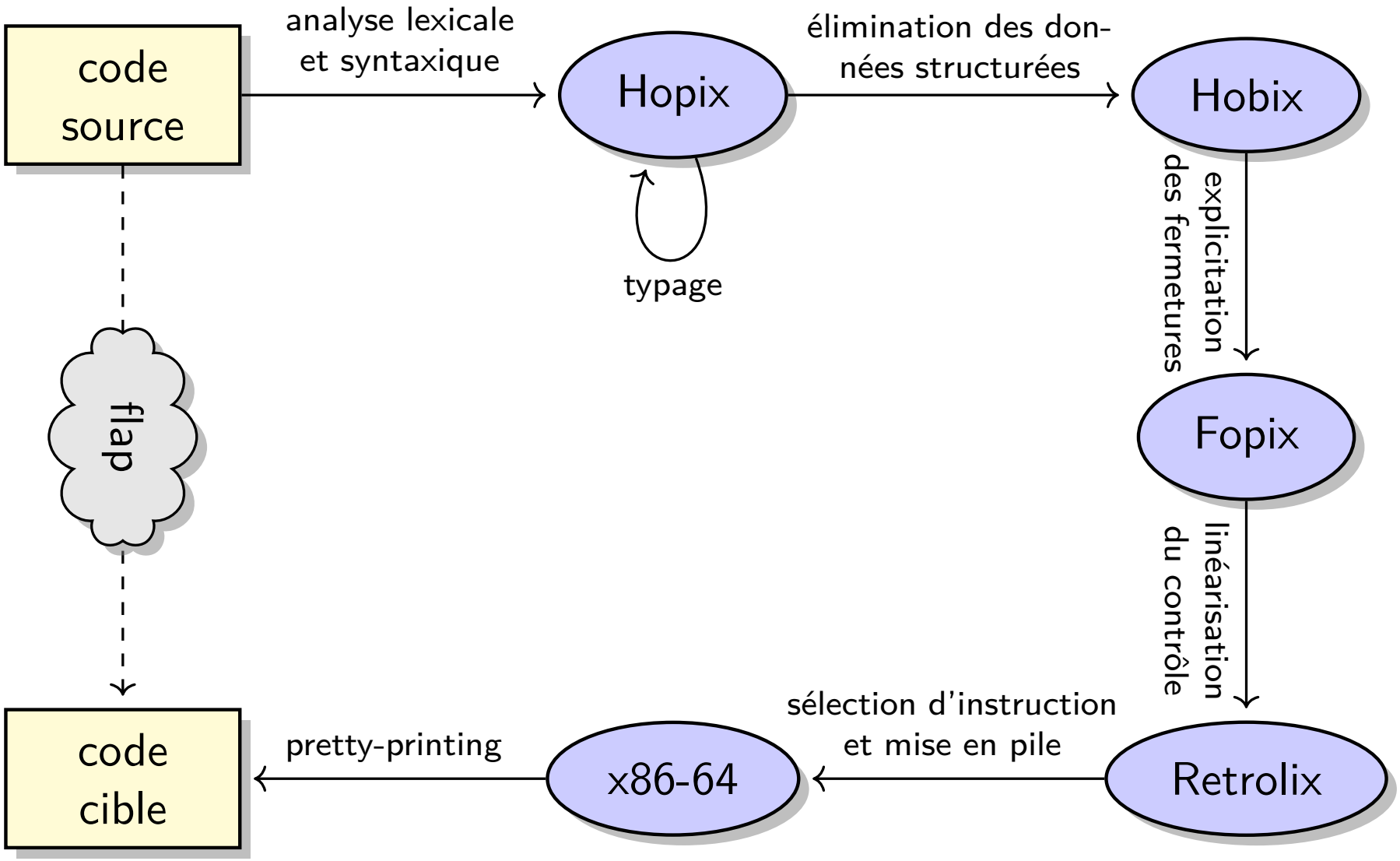
Adrien Guatto

Master 1 Informatique
2022–2023

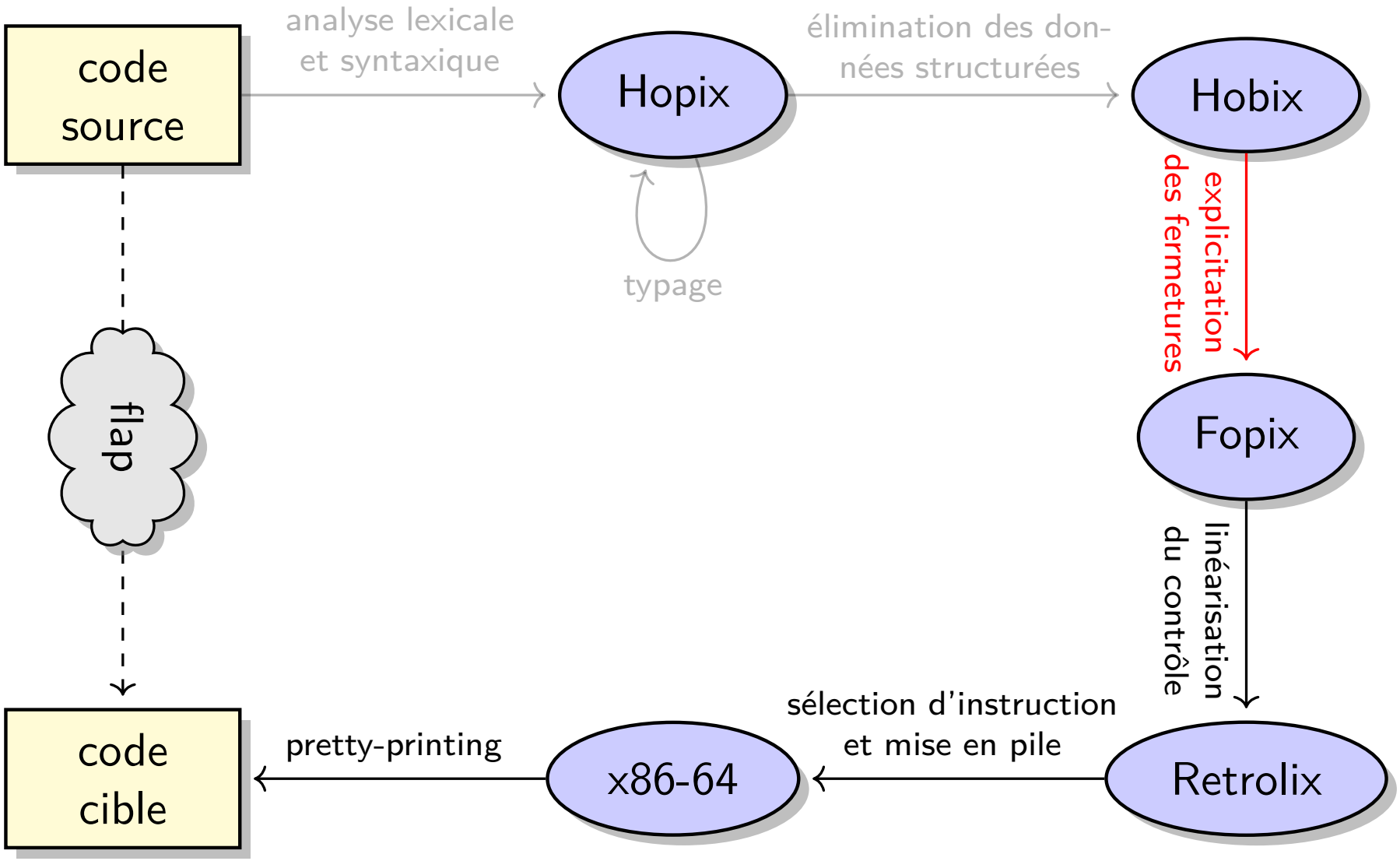
Plan

Plan

Le flot de compilation de flap



Le flot de compilation de flap



Le langage Fopix

```
type program = definition list

and definition =
| DefineValue      of          identifier * expression
| DefineFunction   of function_identifier * formals * expression
| ExternalFunction of function_identifier * int

and expression =
| Literal of literal
| Variable of identifier
| Define of identifier * expression * expression
| FunCall of function_identifier * expression list
| UnknownFunCall of expression * expression list
| While of expression * expression
| IfThenElse of expression * expression * expression
| Switch of expression * expression option array * expression option

and formals = identifier list
```

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;
- de même pour les définitions de valeurs et de fonctions ;

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;
- de même pour les définitions de valeurs et de fonctions ;
- la manipulation de bloc n'apparaît plus explicitement dans la syntaxe.

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;
- de même pour les définitions de valeurs et de fonctions ;
- la manipulation de bloc n'apparaît plus explicitement dans la syntaxe.

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;
- de même pour les définitions de valeurs et de fonctions ;
- la manipulation de bloc n'apparaît plus explicitement dans la syntaxe.

N.B. : on peut toujours appliquer une fonction inconnue (`UnknownFunCall`).

De Hobix à Fopix

Question

Quelles sont les similarités et différences entre Hobix et Fopix ?

Fopix est identique à Hobix, aux différences suivantes près :

- les définitions locales de fonction ont disparu ;
- de même que les fonctions anonymes ;
- les identifiants de valeurs et de fonctions sont maintenant distincts ;
- de même pour les définitions de valeurs et de fonctions ;
- la manipulation de bloc n'apparaît plus explicitement dans la syntaxe.

N.B. : on peut toujours appliquer une fonction inconnue (`UnknownFunCall`).

Moralité

Fopix n'a pas de valeurs fonctionnelles mais des pointeurs de code.

Un langage avec valeurs fonctionnelles : OCaml

```
let ( ** ) g f x = g (f x)

let f g = let g' = g ** ((+) 1) in g' ** g'

let k y = f (fun x -> x + y)

let o = k 2 3
```

Un langage avec valeurs fonctionnelles : OCaml

```
let ( ** ) g f x = g (f x)

let f g = let g' = g ** ((+) 1) in g' ** g'

let k y = f (fun x -> x + y)

let o = k 2 3
```

- Les fonctions sont des valeurs de première classe.
  - On peut les prendre en paramètre et renvoyer comme résultats.
- On peut les introduire par une définition ou anonymement (via **fun**).
- La portée est lexicale, cf. le cours sur l'interprétation.

Un langage avec pointeurs de code : C

```c
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);

void *thread_body(void *arg) { printf("Hello from thread!\n");
                               return NULL; }

int main(int argc, char** argv) {
  pthread_t th; pthread_create(&th, NULL, &thread_body, NULL);
  pthread_join(th, NULL); return 0; }
```

Un langage avec pointeurs de code : C

```c
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);

void *thread_body(void *arg) { printf("Hello from thread!\n");
                               return NULL; }

int main(int argc, char** argv) {
  pthread_t th; pthread_create(&th, NULL, &thread_body, NULL);
  pthread_join(th, NULL); return 0; }
```

- Les pointeurs de fonctions sont des valeurs de première classe.
- Il est impossible de créer une nouvelle fonction.
  - (Sauf bidouilles extrêmes hors langage, e.g., produire du code binaire.)
- Toutes les fonctions ont un nom. Jamais d'imbrication de portée.

Le module `HobixToFopix`

Résumé de la tâche de la passe

- Réduire les valeurs fonctionnelles aux pointeurs de code.
- (Traduire les constructions manipulant les blocs en appels externes.)

Le module `HobixToFopix`

Résumé de la tâche de la passe

- Réduire les valeurs fonctionnelles aux pointeurs de code.
- (Traduire les constructions manipulant les blocs en appels externes.)

Vous avez déjà vu comment implémenter les valeurs fonctionnelles dans un monde qui en est dépourvu...

Le module `HobixToFopix`

Résumé de la tâche de la passe

- Réduire les valeurs fonctionnelles aux pointeurs de code.
- (Traduire les constructions manipulant les blocs en appels externes.)

Vous avez déjà vu comment implémenter les valeurs fonctionnelles dans un monde qui en est dépourvu... C'est le retour des fermetures !

Le module `HobixToFopix`

Résumé de la tâche de la passe

- Réduire les valeurs fonctionnelles aux pointeurs de code.
- (Traduire les constructions manipulant les blocs en appels externes.)

Vous avez déjà vu comment implémenter les valeurs fonctionnelles dans un monde qui en est dépourvu... C'est le retour des fermetures !

L'explicitation des fermetures (ou *closure conversion*)

- On va utiliser la même technique que dans les interprètes.
- Le code généré contient des primitives de manipulation de fermetures.

Le module `HobixToFopix`

Résumé de la tâche de la passe

- Réduire les valeurs fonctionnelles aux pointeurs de code.
- (Traduire les constructions manipulant les blocs en appels externes.)

Vous avez déjà vu comment implémenter les valeurs fonctionnelles dans un monde qui en est dépourvu... C'est le retour des fermetures !

L'explicitation des fermetures (ou *closure conversion*)

- On va utiliser la même technique que dans les interprètes.
- Le code généré contient des primitives de manipulation de fermetures.

Remarque en passant

D'un point de vue très abstrait, le rôle d'un compilateur est donc de *spécialiser* l'interprète générique sur un programme source donné.

Plan

Plan

L'idée générale

Question

Pouvez-vous rappeler ce qu'est une fermeture ?

L'idée générale

Question

Pouvez-vous rappeler ce qu'est une fermeture ?

Abstraitement, une *fermeture* est une paire d'une référence vers du code un environnement qui donne la valeur de ses variables libres. Concrètement ?

L'idée générale

Question

Pouvez-vous rappeler ce qu'est une fermeture ?

Abstraitement, une *fermeture* est une paire d'une référence vers du code un environnement qui donne la valeur de ses variables libres. Concrètement ?

Contexte	**Référence vers le code**	**Environnement**
Évaluateur	Arbre de syntaxe abstraite	Dictionnaire identifiant/valeur
Code compilé	Pointeur de code	Bloc de taille fixe

L'idée générale

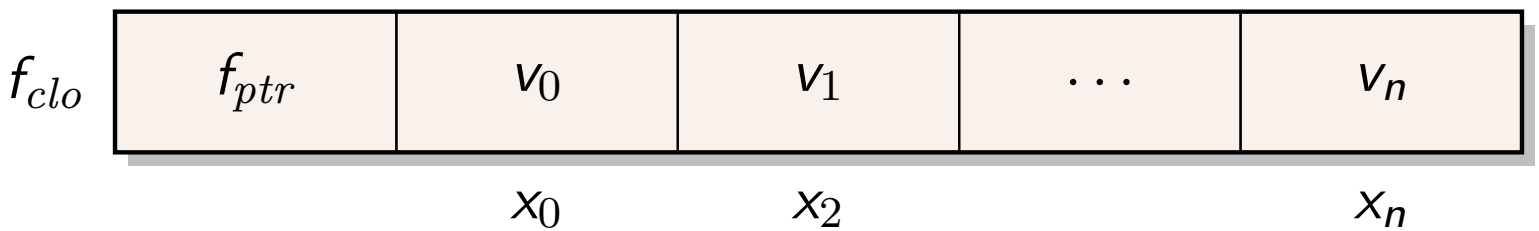
Question

Pouvez-vous rappeler ce qu'est une fermeture ?

Abstraitement, une *fermeture* est une paire d'une référence vers du code un environnement qui donne la valeur de ses variables libres. Concrètement ?

Contexte	**Référence vers le code**	**Environnement**
Évaluateur	Arbre de syntaxe abstraite	Dictionnaire identifiant/valeur
Code compilé	Pointeur de code	Bloc de taille fixe

Une fonction f avec n variables libres $\Leftrightarrow$ un bloc de longueur $n+1$.



Un premier exemple de traduction des abstractions

Comment traduire le code OCaml ci-dessous en code C avec fermetures ?

```
let f0 z = let y = z * 2 in fun x -> x + y + z
```

On crée la fermeture puis extrait, nomme et modifie la fonction anonyme.

Un premier exemple de traduction des abstractions

Comment traduire le code OCaml ci-dessous en code C avec fermetures ?

```
let f0 z = let y = z * 2 in fun x -> x + y + z
```

On crée la fermeture puis extrait, nomme et modifie la fonction anonyme.

```
value f0(value z) {
  block_t *clo = allocate_block(3);
  int y = z * 2;
  clo[0] = &f0_anon0; clo[1] = y; clo[2] = z;
  return clo;
}
```

Un premier exemple de traduction des abstractions

Comment traduire le code OCaml ci-dessous en code C avec fermetures ?

```
let f0 z = let y = z * 2 in fun x -> x + y + z
```

On crée la fermeture puis extrait, nomme et modifie la fonction anonyme.

```
value f0(value z) {
  block_t *clo = allocate_block(3);
  int y = z * 2;
  clo[0] = &f0_anon0; clo[1] = y; clo[2] = z;
  return clo;
}
value f0_anon0(block_t *this, value x) {
  return x + this[1] + this[2];
  /* Une alternative qui évite d'avoir à substituer.
     int y = this[1], z = this[2]; return x + y + z; */
}
```

Un deuxième exemple de traduction des abstractions

```
let f1 y =
  let g = fun x -> x + y in
  let h = fun x -> 2 * x in
  if y > 0 then g else h
```

Un deuxième exemple de traduction des abstractions

```
let f1 y =
  let g = fun x -> x + y in
  let h = fun x -> 2 * x in
  if y > 0 then g else h
```

```
int f1_anon0(block_t *this, int x) { return x + this[1]; }
int f1_anon1(block_t *this, int x) { return 2 * x; }
block_t *f1(value y) {
  block_t *clo;
  if (y > 0) {
    clo = allocate_block(2); clo[0] = &f0_anon0; clo[1] = y;
  } else {
    clo = allocate_block(1); clo[0] = &f0_anon1;
  }
  return clo;
}
```

Traduction des applications

Comment traduire le code OCaml ci-dessous sans faire d'hypothèse sur `f` ?

```
let f2 x = let f = f0 4 in f x
```

On sait cependant que `f0` renvoie une fermeture.

Traduction des applications

Comment traduire le code OCaml ci-dessous sans faire d'hypothèse sur `f` ?

```
let f2 x = let f = f0 4 in f x
```

On sait cependant que `f0` renvoie une fermeture.

```
void f2(int x) {
  block_t *clo = f0(4);
  return clo[0](clo, x);
}
```

Traduction des applications

Comment traduire le code OCaml ci-dessous sans faire d'hypothèse sur `f` ?

```
let f2 x = let f = f0 4 in f x
```

On sait cependant que `f0` renvoie une fermeture.

```
void f2(int x) {
  block_t *clo = f0(4);
  return clo[0](clo, x);
}
// Une autre version, avec un peu d'abstraction.
value apply1(block_t *this, value v) { return this[0](this, v); }

void f2(int x) {
  return apply1(f0(4), x);
}
```

Traitement des fonctions de *top-level*

Dans la traduction précédente, nous avons été très malins !

```
let f2 x = let f = f0 4 in f x
```

```
void f2(int x) { return apply1(f0(4), x); }
```

Question

Pourquoi avons-nous pu traduire si simplement l'application `f0 4` ?

Traitement des fonctions de *top-level*

Dans la traduction précédente, nous avons été très malins !

```
let f2 x = let f = f0 4 in f x
```

```
void f2(int x) { return apply1(f0(4), x); }
```

Question

Pourquoi avons-nous pu traduire si simplement l'application `f0 4` ?

Parce que `f0` est une fonction dite de *top-level*, et donc moralement close[1], et qu'elle est ici *complètement appliquée*. On a optimisé !

1. Ses seules variables libres sont elles-aussi définies à la surface du fichier !

Traitement des fonctions de *top-level*

Dans la traduction précédente, nous avons été très malins !

```
let f2 x = let f = f0 4 in f x
```

```
void f2(int x) { return apply1(f0(4), x); }
```

Question

Pourquoi avons-nous pu traduire si simplement l'application `f0 4` ?

Parce que `f0` est une fonction dite de *top-level*, et donc moralement close[1], et qu'elle est ici *complètement appliquée*. On a optimisé !

Optimiser rend les choses plus subtiles

```
let f3 g x = let f = g 4 in f x
let f2' = f3 f0 (* ATTENTION, f3 traduit attend une fermeture ! *)
```

C'est une bonne idée de ne pas optimiser dans un premier temps.

1. Ses seules variables libres sont elles-aussi définies à la surface du fichier !

La traduction des abstractions (1/2)

On représente toutes les valeurs fonctionnelles des fermetures :

1. les fonctions anonymes,
2. les définitions de valeurs fonctionnelles, y compris *top-level*.

On accumule les définitions Fopix extraites dans un environnement muable.

La traduction des abstractions (1/2)

On représente toutes les valeurs fonctionnelles des fermetures :

1. les fonctions anonymes,
2. les définitions de valeurs fonctionnelles, y compris *top-level*.

On accumule les définitions Fopix extraites dans un environnement muable.

```
val outline_definition : environment -> HobixAST.definition -> unit
val outlined_defs : environment -> HobixAST.definition list
```

La traduction des abstractions (1/2)

On représente toutes les valeurs fonctionnelles des fermetures :

1. les fonctions anonymes,
2. les définitions de valeurs fonctionnelles, y compris *top-level*.

On accumule les définitions Fopix extraites dans un environnement muable.

```
val outline_definition : environment -> HobixAST.definition -> unit
val outlined_defs : environment -> HobixAST.definition list

let rec expression env = function ...
  | HobixAST.Apply (e, actuals) ->
    let open Build in
    let* x = expression env e in
    apply (read_block x 0) (x :: List.map (expression env) actuals)
  | HobixAST.Fun (xs, e) ->
    let free = free_variables e in
    let f, def = name_and_close_fun xs free e in
    outline_function env def;
    allocate_closure f free
```

La traduction des abstractions (2/2)

```
let fold_right_i f xs a =
  snd (List.fold_right (fun x (i, a) -> i + 1, f i x a) xs (0, a))
```

La traduction des abstractions (2/2)

```
let fold_right_i f xs a =
  snd (List.fold_right (fun x (i, a) -> i + 1, f i x a) xs (0, a))


let name_and_close_fun formals free body =
  let open Build in
  let f = fresh_fun_id () and clo_p = fresh_id () in
  f, DefineFunction (f, clo_p :: formals,
       fold_right_i
         (fun i x body ->
           let_ x (read_block clo_p (int (i + 1))) body) free body)
```

La traduction des abstractions (2/2)

```
let fold_right_i f xs a =
  snd (List.fold_right (fun x (i, a) -> i + 1, f i x a) xs (0, a))

let name_and_close_fun formals free body =
  let open Build in
  let f = fresh_fun_id () and clo_p = fresh_id () in
  f, DefineFunction (f, clo_p :: formals,
       fold_right_i
         (fun i x body ->
           let_ x (read_block clo_p (int (i + 1))) body) free body)

let allocate_closure f free =
  let open Build in
  let* b = allocate_block (1 + List.length free) in
  let* _ = write_block b (int 0) f in
  fold_right_i
    (fun i x body -> let* _ = write_block b (int (i + 1)) x in body)
    free b
```

Plan

Et les définitions de fonctions récursives ?

```
let rec repeat n f =
  let rec aux n = if n > 0 then (f n; aux (n - 1)) in
  aux n
```

Et les définitions de fonctions récursives ?

```
let rec repeat n f =
  let rec aux n = if n > 0 then (f n; aux (n - 1)) in
  aux n
```

```
void aux_anon0(block_t *this, value n) {
  if (n > 0) {
    apply1(this[1], n);           /* f n */
    apply1(this[2], n - 1);       /* aux (n - 1) */
  }
}
void repeat(value n, block_t *f) {
  block_t *aux = allocate_block(3);
  aux[0] = &aux_anon0;
  aux[1] = f;
  aux[2] = &aux;                          /* (cycle !) */
  apply1(aux, n);                         /* aux n */
}
```

Le cas particulier de la récursion simple

```
let rec repeat n f =
  let rec aux n = if n > 0 then (f n; aux (n - 1)) in
  aux n
```

```
void aux_anon0(block_t *this, value n) {
  if (n > 0) {
    apply1(this[1], n);          /* f n */
    apply1(this, n - 1);         /* aux (n - 1) */
  }
}
void repeat(value n, block_t *f) {
  block_t *aux = allocate_block(3);
  aux[0] = &aux_anon0;
  aux[1] = f;
  apply1(aux, n);                /* aux n */
}
```

Et la récursion mutuelle ?

```
let repeat_alt n f =
  let rec odd k = if k >= 0 then (f true; even (k - 1))
     and even k = if k >= 0 then (f false; odd (k - 1))
  in
  if n mod 2 = 0 then even n else odd n
```

Question

Comment traduire ce code vers C en introduisant des fermetures ?

Et la récursion mutuelle ?

```
let repeat_alt n f =
  let rec odd k = if k >= 0 then (f true; even (k - 1))
     and even k = if k >= 0 then (f false; odd (k - 1))
  in
  if n mod 2 = 0 then even n else odd n
```

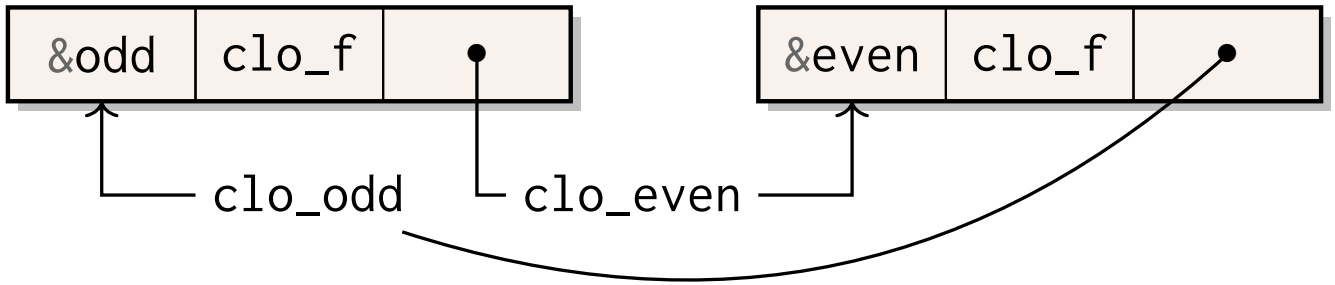
Question

Comment traduire ce code vers C en introduisant des fermetures ?

Il existe deux grandes approches :

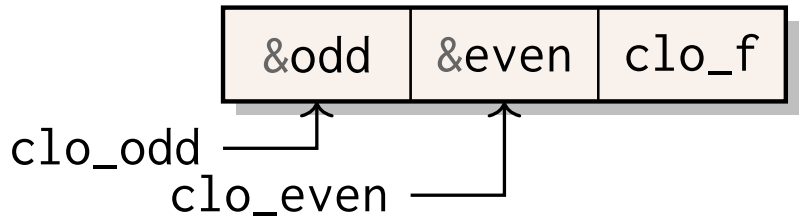
- une fermeture pour `odd`, une pour `even` ;
- une seule fermeture partagée entre `odd` et `even`.

La récursion mutuelle avec fermetures distinctes




```
void odd(block_t *this, value k) {
  if (k >= 0) {
    apply1(this[1], 1);          /* f true */
    apply1(this[2], k - 1);      /* even (k - 1) */
  }
}
void even(block_t *this, value k) {
  if (k >= 0) {
    apply1(this[1], 0);          /* f false */
    apply1(this[2], k - 1);      /* odd (k - 1) */
  }
}
void repeat_alt(value n, block_t *clo_f) {
  block_t *clo_odd = allocate_block(3), *clo_even = allocate_block(3);
  clo_odd[0] = &odd; clo_odd[1] = clo_f; clo_odd[2] = clo_even;
  clo_even[0] = &even; clo_even[1] = clo_f; clo_even[2] = clo_odd;
  if (n % 2 == 0) { apply1(clo_even, n); } else { apply1(clo_odd, n); }
}
```

La récursion mutuelle avec fermeture partagée




```
void odd(block_t *this, value k) {
  if (k >= 0) {
    apply1(this[2], 1);          /* f true */
    apply1(&this[1], k - 1);      /* even (k - 1) */
  }
}
void even(block_t *this, value k) {
  if (k >= 0) {
    apply1(this[1], 0);          /* f false */
    apply1(&this[-1], k - 1);    /* odd (k - 1) */
  }
}
void repeat_alt(value n, block_t *clo_f) {
  block_t *clo = allocate_block(3);
  clo[0] = &odd; clo[1] = &even; clo[2] = clo_f;
  block_t *clo_even = &clo[0], *clo_odd = &clo[1];
  if (n % 2 == 0) { apply1(clo_even, n); } else { apply1(clo_odd, n); }
}
```

Plan

Les fonctions à plusieurs arguments

Langages fonctionnels et arité

- En Hopix/OCaml/SML, toutes les fonctions sont unaires.
  - Hopix/OCaml : les fonctions n-aires sont simulées par curryfication.
  - SML : les fonctions n-aires sont simulées via les n-uplets.
- En Hobix/Fopix/C/Scheme, les fonctions sont n-aires (n quelconque).

Les fonctions à plusieurs arguments

Langages fonctionnels et arité

- En Hopix/OCaml/SML, toutes les fonctions sont unaires.
  - Hopix/OCaml : les fonctions n-aires sont simulées par curryfication.
  - SML : les fonctions n-aires sont simulées via les n-uplets.
- En Hobix/Fopix/C/Scheme, les fonctions sont n-aires (n quelconque).

Notre traduction est fausse

Elle traite mal les fonctions n-aires (avec $n > 1$) appliquées partiellement.

La racine du problème

Lisons le code qui suit comme du code Hobix, où `f` est réellement binaire.

```
let f x y = x + y
let g h = h 2
let k = g (f 1)
```

Question

Que donne l'explicitation des fermetures sur ce code ?

La racine du problème

Lisons le code qui suit comme du code Hobix, où `f` est réellement binaire.

```
let f x y = x + y
let g h = h 2
let k = g (f 1)
```

Question

Que donne l'explicitation des fermetures sur ce code ?

```
value f_anon0(block_t *this, value x, value y) { return x + y; }
block_t *g_anon0(block_t *this, block_t *h) { apply1(h, 2); }
block_t *f, *g; value k;
void initialize() {
  f = allocate_block(1); f[0] = &f_anon0;
  g = allocate_block(1); g[0] = &g_anon0;
  k = apply1(g, apply1(f, 1));
}
```

Deux applications sur trois sont fausses, moralement mal typées en C.

Une remarque sur Hopix vers Hobix

Est-ce un problème ?

Si on a traduit les fonctions unaires de Hopix en fonctions unaires Hobix, ce problème ne se pose pas car toutes les applications sont totales.

Autrement dit, notre traduction n'est valide que sur le fragment de Hobix qui est l'image de la traduction de Hopix.

Une remarque sur Hopix vers Hobix

Est-ce un problème ?

Si on a traduit les fonctions unaires de Hopix en fonctions unaires Hobix, ce problème ne se pose pas car toutes les applications sont totales.

Autrement dit, notre traduction n'est valide que sur le fragment de Hobix qui est l'image de la traduction de Hopix.

Pourquoi pourrait-on vouloir une solution générale ?

- Des optimisations de Hobix pourraient sortir de ce fragment.
- On peut imaginer une traduction de Hopix plus efficace.

Détaillons ce dernier point.

Pourquoi un traitement spécial des fonctions *n*-aires ?

```
let f1 x y = 2 * x + y
let f2 x =
  if Sys.file_exists (Printf.sprintf "toto%d.txt" x)
  then fun y -> 2 * x + y else fun y -> 0
let a = f1 1 2
let b = f2 1 2
```

Avec la solution actuelle, le premier appel coûte aussi cher que le second.

La solution employée dans OCaml

- Chaque clôture enregistre le nombre de paramètres formels attendus.
- À l'application, on compare au nombre d'arguments effectifs.
- S'ils sont égaux, on évite de créer des fermetures intermédiaires !

Plan

Conclusion

L'explicitation des fermetures

- Essentiel à la plupart des compilateurs de langages fonctionnels.
  - Quasiment tous ceux que je connais. Exception : MLton.
- Plutôt simple dans sa version de base, peut devenir complexe.
  - Par exemple, si on veut typer précisément les étapes intermédiaires (Minamide, Morrisett et Harper 1996).
  - Beaucoup de travail sur la représentation des environnements jusqu'à début 2000 (Shao et Appel 2000, par exemple).

L'explication des fermetures est le sujet du prochain jalon.

Bibliographie


Minamide, Yasuhiko, Greg Morrisett et Robert Harper (1996). "Typed closure conversion". In : *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*. POPL '96. ACM Press. DOI : 10.1145/237721.237791. URL : https://www.cs.cmu.edu/~rwh/papers/closures/popl96.pdf.


Shao, Zhong et Andrew W. Appel (jan. 2000). "Efficient and Safe-for-Space Closure Conversion". In : *ACM Trans. Program. Lang. Syst.* 22.1, p. 129-161. ISSN : 0164-0925. DOI : 10.1145/345099.345125. URL : https://www.classes.cs.uchicago.edu/archive/2011/spring/22620-1/papers/closure-conversion.pdf.