Université de Montréal

Phase d'optimisation

Le langage d'entrée est généralement identique à celui de sortie

Utilise souvent la synergie entre optimisations

Ordre pas forcément évident; application réitérée

Éliminer le coût des abstractions

- Optimiser le code qui vient du compilateur lui-même

Laisser le programmeur optimiser son code

- Pourtant code idiot n'implique pas programmeur idiot

Souvent besoin d'une *analyse* antérieure

Université de Montréal

Phase d'analyse

Renvoie une structure auxiliaire ou une nouvelle représentation du code

Extraire l'information nécessaire à des phases ultérieures

Exemples:

- Trouver tous les usages de chaque variable/fonction
- Déterminer les variables *vivantes* à chaque point
- Trouver les dépendances entre opérations
- Trouver l'ensemble des sous-types possibles
- Calculer l'ensemble des valeurs possibles

Résultat potentiellement invalidé par chaque optimisation

Université de Montréal

Optimisations simples

Élimination de code mort

Propagation de constantes

Constant folding

Propagation de variables

Strength reduction

Élimination de code mort

Éliminer le code inaccessible

- Fonctions inutilisées
- Code après quelque sorte de `goto`
- Code après un appel de fonction sans fin

Éliminer les opérations inutilisées

- `let x = y + z in 5`
- Attention aux effets de bord

Besoin d'analyse, e.g. tous usages de chaque var/function

Université de Montréal

Élimination de vars mortes (analyze)

```
deadvars : Exp -> (Exp * Set Id)
deadvars e = case e
  | Var x => (e, set_singleton x)
  | Num _ => (e, set_empty)
  | Fun x e
    => let (e', xs) = deadvars e
       in (Fun x e', set_remove x xs)
  | App e1 e2
    => let (e1', xs1) = deadvars e1
           (e2', xs2) = deadvars e2
       in (App e1' e2', set_union xs1 xs2)
```

Université de Montréal

Élimination de vars mortes (optimisation)

```
| Let x e1 e2
  => let (e2', xs2) = deadvars e2
     if (set_member x xs2 || !pure e1) then
        let (e1', xs1) = deadvars e1
        in (Let x e1' e2',
            set_union (set_remove x xs2) xs1)
     else
        (e2', xs2)
```

Pas d'élimination d'arguments inutilisés

Université de Montréal

Constant folding

Pré-exécution de code lors de la compilation

Cas limité d'*évaluation partielle*

Exemples:

```
                    e + 0  ⇝  e
                    3 + 4  ⇝  7
            (x0, x1, x2).1  ⇝  x1
if true then e1 else e2  ⇝  e1
```

Université de Montréal

Constant folding (exemple)

```
cstfold e = case e
| (Var _ | Num _) => e
| App e1 e2 => App (cstfold e1) (cstfold e2)
| Prim (Add, [e, Num 0]) => e
| Prim (Add, [Num 0, e]) => e
| Prim (Add, [Num n1, Num n2]) => Num (n1 + n2)
| Prim (Get, [Tuple t, Num i]) => List.nth i t
 ...
```

Université de Montréal

Strength reduction

Remplace une opération par une autre plus simple

Exemples:

```
x * 2  ⇝  x + x

e * 16  ⇝  e << 4

(fun x => e1) e2  ⇝  let x = e2 in e1
```

Ou encore, dans les boucles:

```
i = i + 1, p + i * 7  ⇝  p' = p' + 7, p'
```

Université de Montréal

Propagation de constantes

Déplace les constantes vers leur point d'usage

- Peu utile en soi
- Expose des opportunités pour d'autres optimisations

Plutôt que constantes, peut propager des expressions simples

- Exemple typique: propagation de variables

Université de Montréal

Propagation de constantes (exemple)

```
cstprop : Map Id Exp -> Exp -> Exp
cstprop c e = case e
| Var x => case map_lookup x ctx
    | some e' => adjust_scope e'
    | none -> e
| App e1 e1 => App (cstprop c e1) (cstprop c e2)
| Num _ => e
| Let x e1 e2
  => let e1' = cstprop c e1
         c' = if not (propagate x e1') then c
              else map_add x e1' c
     in Let x e1' (cstprop c' e2)
```

Université de Montréal

Inlining

Forme de propagation de constante

Copie le code d'une fonction à un point d'appel

- Éviter le coût de l'appel
- Spécialiser le code de la fonction pour cet appel

Pas utilisable pour les appels indirects

Très utilisé par d'autres optimisations

- Uncurrying
- Élimination des arguments morts

Risque d'explosion de code!

Où faire du inlining

Certaines (petites) fonctions devraient être *inlined* partout

- Attention à la récursion
- Même petites, l'augmentation de code peut-être susbtantielle

Analyser la fonction pour trouver les opportunités de spécialisation

- Un des arguments est appelé (fonction d'ordre supérieur)
- Une valeur d'argument permet d'éliminer un partie du code
- L'opportunité est dans une boucle

Université de Montréal

*Exemple d'*inlining

```
map : (a -> b) -> List a -> List b
map f xs = case xs
  | [] => []
  | x :: xs => f x :: map f xs
```

Cas utiles:

- `map somefun []`
- `map (fun x => x + 1) l`

Attention au *peeling*

Spécialisation de fonction

Création d'un certain nombre de copies *spécialisées*

- Pour une valeur particulière d'argument
- Avec des arguments passés de manière non-standard

Remplacement de *inlining*

Meilleur contrôle de la taille du code

Versions à usage unique finissent *inlined*