Function inlining and constant folding

- Perform all constant computations at compile time
- Substitution e[a / v]:
  - *substitute a for v in e*
  - *a substitution is a (atom SymbolTable.t)*
- Have to be careful not to shadow definitions

```
let v1 = 1 in
let v2 = v1 in
v1 <- 2;
f(v2)
```

```
let v1 = 1 in
v1 <- 2;
f(v1)
```

Inlining

- FIR: substitution is always valid
- First, characterize values that may be inlined

```
(*
 * Available expressions.
 *)
type avail =
   AvailVar of var
 | AvailAtom of atom
 | AvailClosure of var * var * avail
 | AvailFunction of var * bool * var list * exp
 | AvailRecord of var

type arecord = avail FieldTable.t

type aenv =
   { aenv_vars : avail SymbolTable.t;
     aenv_records : arecord SymbolTable.t
   }
```

“Available” expressions

- The aenv contains an *available* expression for each var v
  - *AvailVar v’: simple case, variable v is available as var v’*
  - *AvailAtom a: variable v can be replaced by a*
  - *AvailClosure (v’, f, arg) variable v is defined as a closure (f, arg), v’ contains its value*
  - *AvailFunction (v’, force, f, args), variable v is a function f(args), v’ contains its value, force is set if inlining should be forced*
  - *AvailRecord v’: variable v is a record defined in the available records as v’*

Inlining LetAtom

- LetAtom can always be inlined

```
and inline_atom_exp aenv pos v a e =
  let pos = string_pos "inline_atom_exp" pos in
  let avail = aenv_lookup_atom aenv a in
  let aenv = aenv_add aenv v avail in
    inline_exp aenv e
```

Constant folding

- Perform constant arithmetic at compile time

```
and inline_unop_exp aenv pos v ty op a e =
  let pos = string_pos "inline_unop_exp" pos in
  let v' = new_symbol v in
  let a = inline_atom aenv a in
  let a' =
    match op, a with
      UMinusIntOp, AtomInt i ->
        AtomInt (-i)
    | UMinusFloatOp, AtomFloat x ->
        AtomFloat (-.x)
    | ...
  in
  let aenv = aenv_add_atom aenv v a' in
  let e = inline_exp aenv e in
    LetUnop (v', ty, op, a, e)
```

Give a new name to the var

Perform constant folding

Inline the rest

Constant folding

- Have to be careful:
  - *is compiler arithmetic the same as run-time?*
    - *Example: OCaml uses 31 bit ints, Java uses 32*
    - *we'll ignore this for now*
  - *what about divide by zero?*
    - *may be ok to flag error at compile time*
    - *if not, do not inline division by 0*

Conditionals

- Conditionals can be evaluated too

```
and inline_if_exp aenv pos a e1 e2 =
  let pos = string_pos "inline_if_exp" pos in
    match inline_atom aenv a with
      AtomBool true ->
        inline_exp aenv e1
     | AtomBool false ->
        inline_exp aenv e2
     | a ->
        let e1 = inline_exp aenv e1 in
        let e2 = inline_exp aenv e2 in
          IfThenElse (a, e1, e2)
```

Function inlining

- Inline “small” functions
- Initial IR usually has very small functions (1-5 instructions)
- Have to be careful to avoid code blowup
  - *Don’t inline recursive functions*
  - *Be careful about inlining in conditionals*

Function “size”

```
(* These don't cost anything *)
 LetVar (_, _, a, e)
| LetAtom (_, _, a, e) ->
    size_exp fenv size e

 (* Cheap expressions *)
| LetSubscript (_, _, _, _, e)
| LetProject (_, _, _, _, e)
| LetArray (_, _, _, _, e) ->
    size_exp fenv (succ size) e
```

Functions: unknown definition

```
and inline_tailcall_exp aenv pos f args =
  let pos = string_pos "inline_tailcall_exp" pos in
  let rec inline aenv f args =
    match aenv_lookup aenv f with
      AvailVar f ->
        TailCall (f, List.map atom_of_avail args)
```

Functions: closure

- Recursively inline the closure

```
| AvailClosure (_, f, arg) ->
    inline aenv f (arg :: args)
```

Function: function def

```
| AvailFunction (f', force, vars, body) ->
    let aenv =
      if force then
        aenv
      else
        aenv_add_id aenv f'
    in
    let aenv = aenv_add_args aenv pos vars args in
      inline_exp aenv body
```

"Forced" functions remain in available set

Otherwise, remove the body

Common-subexpression elimination

- Some computations may be performed multiple times (without side-effects)
  - *May be introduced by programmer (perhaps unintentionally)*
  - *May be introduced during IR generation*
- Fold them together

```
while(i < 10) {
    a[j + i] = a[j + i] + a[i];
    i++;
}
```

CSE

```
let loop (j, i) =
   if i < 10 then
      let v1 = j + i in
      let v2 = a[v1] in
      let v3 = a[i] in
      let v4 = v2 + v3 in
      let v5 = j + i in
         a[v5] <- v4;
         let i = i + 1 in
            loop(j, i)
   else
      break()
```

```
let loop (j, i) =
   if i < 10 then
      let v1 = j + i in
      let v2 = a[v1] in
      let v3 = a[i] in
      let v4 = v2 + v3 in
         a[v1] <- v4;
         let i = i + 1 in
            loop(j, i)
   else
      break()
```

Structuring CSE

- FIR invariants
  - *Each variable has one def*
  - *Let-definitions have no side-effects*
  - *Two expressions compute the same value if they have the same code*
- Simple CSE
  - *Fold all identical let-definitions*
- Problems:
  - *Sub-optimal: are "x + y" and "y + x" the same?*
  - *Memory dereferences: memory ops are not functional*

Operations that can be folded

- LetAtom (can be totally eliminated)
- LetUnop
- LetBinop
- Address-of (LetAddrOfSubscript, LetAddrOfProject)
- LetClosure

CSE Algorithm

- Keep a table of full-expanded expression trees for each let-definition
  - *expr_table: exp -> var*
- For each foldable let-definition
  - *Calculate its full-expanded expression tree*
  - *If expression tree is in the table, use the existing value in a LetAtom definition*
  - *If not, add the let definition to the table, keep existing code*

Expression trees

```
type cse =
    CseAtom of atom
  | CseUnop of unop * cse
  | CseBinop of binop * cse * cse
  | CseClosure of var * cse list
```

Computing expression trees

```
let v1 = i + j in             (i + j) -> v1
let v2 = v1 + k in            ((i + j) + k) -> v2
let v3 = j + k in             (j + k) -> v3
let v4 = i + v3 in            (i + (j + k)) -> v4  (and v2?)
...
```

CSE

- Data structure: (cse -> var) Table
  - *In practice, naïve CSE is pretty successful*
  - *Use table equality to generalize folding*

CSE equality

```
let rec compare_exps e1 e2 =
   match e1, e2 with
      CseAtom a1, CseAtom a2 ->
         compare_atoms a1 a2
    | CseUnop (op1, e1), CseUnop (op2, e2) ->
         let i = Pervasives.compare op1 op2 in
            if i = 0 then
               compare_exps e1 e2
            else
               i
```

CSE Expression table

- Use a normal table with expression comparison

```
module CseCompare =
struct
   type elt = cse
   let compare = compare_exps
end

module CseTable = Mc_map.MakeTable (CseCompare)
```

CSE Algorithm

- When a let-definition is encountered
  - *Compute expression tree*
  - *Look it up in CseTable*
  - *If Not_found*
    - *Keep the let-definition, add it to CseTable*
  - *Otherwise*
    - *let-definition becomes LetAtom*

CSE Binop case

```
and cse_binop_expr venv pos v ty op a1 a2 e =
   let exp = CseBinop (op, CseAtom a1, CseAtom a2) in
      match venv_lookup_exp venv pos exp with
         Some a, exp ->
            let venv = venv_add_var venv v exp in
               LetAtom (v, ty, a, cse_expr venv e)
       | None, exp ->
            let venv = venv_add_var venv v exp in
               LetBinop (v, ty, op, a1, a2, cse_expr venv e)
```

CSE Lookup

- Fully-expand the expression
- Look it up

```
let venv_lookup_exp venv pos exp =
   let exp = expand_exp venv pos exp in
   let a =
      try Some (AtomVar (CseTable.find venv.cse_exps exp)) with
         Not_found ->
            None
   in
      a, exp
```

CSE Recursive expansion

- Variables are already full-expanded

```
let rec expand_exp venv pos e =
   match e with
      CseAtom (AtomVar v) ->
         venv_lookup_var venv pos v
    | CseAtom _ ->
         e
    | CseUnop (op, e) ->
         CseUnop (op, expand_exp venv pos e)
    | CseBinop (op, e1, e2) ->
         CseBinop (op, expand_exp venv pos e1, expand_exp venv pos e2)
    | ...
```

Classes, Interfaces, and Records

- In FIR, we reduce classes and interfaces to records

```
type record_class =
   RecordFrame
 | RecordClass
 | RecordMethods
 | RecordObject
 | RecordVObject

type ty =
   ...
   (* TyRecord is a collection of fields *)
 | TyRecord of record_class * field_types
   ...

(* FieldTable is the nonoverloaded FieldMTable *)
and field_types = ty FieldTable.t
```

Records

- The “record_class” describes what kind of record
  - *RecordFrame: introduced during closure conversion*
  - *RecordClass: a vma record for a class (includes a “names” record)*
  - *RecordMethods: a vma record for an interface (just a collection of methods)*
  - *RecordObject: an object record*
    - *The first field points to a RecordClass*
    - *The rest of the fields are the object’s fields*
  - *RecordVObject:*
    - *A two-element record (vma + this) for a vobject*

Other record info

- FieldTable
  - *Just like a FieldMTable, but it does not have external names (FIR does not use external names)*

Upcasts

- In addition to VMA, store a "name" table

```
class A { ... }

interface C { ... }
interface D { ... }

class B
extends A
implements C, D
{ ... }
```


FjObject	nil
A	nil
B	nil
C	b_c_vma
D	b_d_vma
nil	nil

Our solution

- Interface objects include an interface VMA, and an object
- Coercions are more expensive (although still constant time), method lookup is constant time

RecordMethods
s_c_vma
c.f
b.g

RecordVObject
VObject(S)
s_c_vma
object

Object C
c_vma
c_fields...

c_vma
b.z
b.g
b.h
c.f

FIR conversion

- CPS and record conversion are performed simultaneously
- To convert a class_info
  - *Allocate the names record*
    - *from the class_parents and class_interfaces*
  - *Allocate the vma record*
    - *From the class_methods*
  - *Define the object record type*
    - *From the class_fields*

Class/interface projections

- This is the only other interesting conversion
- When projecting an object field
  - *Just project it*
- When projecting a method
  - *Double projection:*
    - *Project the vma*
    - *Project the method*
- When projecting a vobject method
  - *Another double projection*
- For LetProjObject
  - *Project the object from the record*