Administrative Notes

Here is the remaining lecture schedule.

4/2 Abstract Interpretation

4/4 Partial Evaluation

4/9 Program Verification

4/11 Subtyping revisited

4/16 No class

4/18 No class

4/23 Java

4/25 Parallel Languages

4/30 Distributed Languages

5/2 Project presentations (3 hours)

Static Analysis

Static analysis is used to:

- Detect program errors, such as type errors.
- Enable program optimizations.

The analyses should:

- Terminate (minimal requirements)
- Be efficient (depends somewhat on problem domain)
- Provide useful information about control flow, data dependencies, etc.
- May require language restrictions, such as strong typing

Type and effect checking use static analysis based on operational reasoning: the type/effect rules look like SOS rules.

Abstract Interpretation

Abstract interpretation: the compiler "executes" the program on an abstract domain of input values to infer things about the values of variables in the program.

- No language restrictions are required.
- Approach uses denotational semantics.
- Traditional analyses, e.g., control flow and data flow, can be cast in this framework

May be used for:

- Strictness analysis in lazy languages [Mycroft], [Peyton-Jones]
- Memory management [Morrisett, Felleisen, and Harper], [Hudak]
- Invariants [Cousot and Cousot]
- Array bounds checking
- Alias analysis [Deutsch]
- Logic language evaluation [Marriott and Sondergaard]

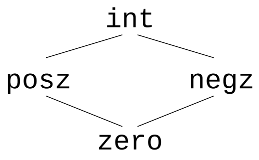
Example: Law of Signs

Consider a very simple expression language:

E $\in$ Expression ::= N | `(+ E1 E2)` | `(* E1 E2)`

Assume we're interested in optimizations based on the sign of a numeric value.

S $\in$ Sign = posz | negz | zero | int



posz = positive $\cup$ 0
negz = negative $\cup$ 0
zero = { 0 }
int = all integers

Where the above sets are not disjoint, but form a lattice by subset inclusion.

Law of Signs

Define a function *sign* that assigns a one of these signs to each expression.
sign: Exp $\rightarrow$ Sign
Approaches:

- It would be correct, but not very precise, to define:

$$\text{sign } [\![E]\!] = \texttt{int} \text{ for all E.}$$

- A completely precise definition would be:

$$\text{sign } [\![E]\!] = (\text{exactsign } \mathcal{E}[\![E]\!] \ )$$

$$\begin{array}{rl} (\text{exactsign} N) = & \textbf{matching} N \\ & \triangleright\, 0 \ : \ \texttt{zero} \\ & \triangleright\, n > 0 \ : \ \texttt{posz} \\ & \triangleright\, n < 0 \ : \ \texttt{negz} \\ & \textbf{endmatching} \end{array}$$

These two definitions represent the extremes of precision.

Abstract Interpretation

In between the two extremes, we can define:

$$\text{sign}[\![E]\!] = \mathcal{E}_{abs}[\![E]\!]$$

$$\begin{aligned} \mathcal{E}_{abs}[\![N]\!] &= (\text{exactsign } N) \\ \mathcal{E}_{abs}[\![(+\ \texttt{E1 E2})]\!] &= (+_{sign}\ \mathcal{E}_{abs}[\![E1]\!]\ \mathcal{E}_{abs}[\![E2]\!]) \\ \mathcal{E}_{abs}[\![(*\ \texttt{E1 E2})]\!] &= (*_{sign}\ \mathcal{E}_{abs}[\![E1]\!]\ \mathcal{E}_{abs}[\![E2]\!]) \end{aligned}$$

Where $*_{sign}$ and $+_{sign}$ are defined as:

$(*_{sign}$ `zero S) = zero`	$(+_{sign}$ `zero S) = S`
$(*_{sign}$ `posz posz) = posz`	$(+_{sign}$ `posz posz) = posz`
$(*_{sign}$ `negz negz) = posz`	$(+_{sign}$ `negz negz) = negz`
$(*_{sign}$ `negz posz) = negz`	$(+_{sign}$ `negz posz) = int`

`if S` $\neq$ `zero:`

$(*_{sign}$ `int S) = int`	$(+_{sign}$ `int S) = int`

Plus the equations based on commutativity.

Abstract Interpretation

An abstraction interpretation function (like sign) maps from the concrete to the abstract domain.

In general, these abstract interpretations are described by *commutative diagrams*.

$$\begin{tikzcd} \texttt{Sign x Sign} \arrow[r] & \texttt{Sign} \arrow[r, phantom, "\text{abstract domain}"] & {} \\ & \rotatebox{90}{$\subseteq$} & \\ \texttt{Int x Int} \arrow[r] \arrow[uu] & \texttt{Int} \arrow[u] \arrow[r, phantom, "\text{concrete domain}"] & {} \end{tikzcd}$$

Without the $\subseteq$ notation on the abstraction edge, such a diagrams represents a homomorphism.

With the $\subseteq$, it represents a semi-homomorphism.

Notes on Abstract Interpretation

The precise meaning of the program may be given by the usual dynamic semantics or the program, or by some interesting property: termination, type, execution time, memory use, etc.

The approximate meaning is a *conservative approximation* of the precise meaning.

- correct – there is a commutative diagram
- conservative – the abstract edge has a subset relation, not equality

In practice, abstract interpretation is difficult over real programming languages, so we will consider some more examples of restricted languages.

Example: First-Order Equations

Consider the following simple language of first-order recursive eqations, FORQ.

P $\in$ Program
E $\in$ Exp
L $\in$ Literal
O $\in$ Primop
F $\in$ FunctionName
I $\in$ Identifier

P ::= `(program (define (F I*)` E$_{body}$`)*` E$_{top}$`)`
E ::= `L | I | (if` E$_{test}$ E$_{then}$ E$_{else}$ `)`
  `| (F` E$_{arg}$`*) | (primop O E*)`
  `| (let ((I E)*)` E$_{body}$`)`

- Procedure definitions are not nested.
- Procedure and function names are separate spaces.
- Procedures are not first class.

Strictness Analysis

Consider a lazy version of FORQ. When is it safe to use eager evaluation (usually more efficient) in place of lazy for a given parameter?

Answer: when the function is strict in

An n-ary function `f` is *strict* in its i^{th} argument if:

$$(\texttt{f}\ E_1 \ldots E_{i-1}\ \perp\ E_{i+1} \ldots E_n) = \perp$$

for all $E_1 \ldots E_{i-1} E_{i+1} \ldots E_n$.

Examples:

- `add3` is strict in `x` but not `y`.
  `(define (add3 x y) (+ x 3))`
- `cand` is strict in `y` but not `x`.
  `(define (cand x y) (if y x y))`
- `h` is strict in both `x` and `y`.
  `(define (h x y) (if (< x 0) y (h (- x 1) y)))`

Strictness Analysis in FORQ

The abstract domain, T, has two sets:

- `unknown`: possibly terminating, given by 1
- `bottom`: definition non-terminating, given by 0

`unknown`
`bottom`

Where `bottom` $\subseteq$ `unknown`.

Assume boolean operatoon AND and OR on T, with 1 (`unknown`) interpreted as true and 0 (`bottom`) as false. The interpretation is:

$$\mathcal{E}[\![L]\!] = 1$$
$$\mathcal{E}[\![I]\!] = I$$
$$\mathcal{E}[\![\texttt{(primop P E1 E2)}\ ]\!] = \mathcal{E}[\![\texttt{E1}]\!] \text{ AND } \mathcal{E}[\![\texttt{E2}]\!]$$
$$\mathcal{E}[\![\texttt{(F E1 E2)}\ ]\!] = \mathcal{E}[\![\texttt{F}]\!]\ \mathcal{E}[\![\texttt{E1}]\!]\ \mathcal{E}[\![\texttt{E2}]\!]$$
$$\mathcal{E}[\![\texttt{(if E1 E2 E3)} = \mathcal{E}[\![E1]\!] \text{ AND } (\mathcal{E}[\![E2]\!] \text{ OR } \mathcal{E}[\![E3]\!])$$

Interpretations of Simple FORQ Expressions

First, we'll look at some non-recursive functions.

`(define (add3 x y) (primop + x 3))`

Then the interpretation of `add3` is:

$$\mathcal{E}[\![\texttt{add3}]\!] = x \text{ AND } 1$$

which means:

$$(\mathcal{E}[\![\texttt{add3}]\!]\ 0\ 1) = 0$$
$$(\mathcal{E}[\![\texttt{add3}]\!]\ 1\ 0) = 1$$

So `add3` is strict in `x` but not in `y` as expected.

Interpretations of Simple FORQ Expressions

The `cand` example also works:

```
(define (cand x y) (if y x y))
```

The interpretation of `cand` is:

$$\mathcal{E}[\![\texttt{cand}]\!] = y \text{ AND } (x \text{ OR } y)$$

which means:

$$(\mathcal{E}[\![\texttt{cand}]\!]\ 0\ 1) = 1$$
$$(\mathcal{E}[\![\texttt{cand}]\!]\ 1\ 0) = 0$$

So `cand` is strict in `y` but not in `x`.

Recursion and Strictness Analysis

How do we determine whether a recursive function is strict in any given argument? Compute a fixed point on the abstract domain.

```
(define (h x y) (if (< x 0) y (h (- x 1) y)))
```

Let `h0,h1,...` be increasingly accurate approximations to the interpretation of `h`, with the first one defined as bottom.

```
(h0 x y) = 0

(h1 x y) = (x AND 1) AND (y OR (h0 (x AND 1) y))
         = (x AND 1) AND (y OR 0)
         = x AND y

(h2 x y) = (x AND 1) AND (y OR (h1 (x AND 1) y))
         = (x AND 1) AND (y OR (x AND y))
         = x AND y        <-- fixed point reached
```

So `h` is strict in both arguments.

Memory Management

Compile time reference counting can be used to eliminate garbage collection overhead.
Consider another purely functional version of FORQ, but this time with vectors.

```
E ::= ... | (make-vector n default)
          | (vector-ref vec index)
          | (vector-update vec index new)
```

A naive implementation would produce a complete copy of a vector whenever the update operation is used. If there is only one reference to a vector, then we may be able to reuse the old space for the new one.

```
(program
  (define (init vec index)
    (if (>= index 100)
        vec
        (init (vector-update vec index index)
              (+ index 1))))
  (init (make-vector 100 -1) 0))
```


Memory Management

Semantic domains for FORQ + Vectors with reference counts.

$$Env = Id \rightarrow Val$$
$$Fenv = Function\text{-}Name \rightarrow Proc$$
$$Proc = Val^* \rightarrow Sto \rightarrow (Val \times Sto)$$
$$Val = Int + Bool + Loc$$
$$Loc = Int$$

$$Sto = Loc \rightarrow Int^* \times RC$$
$$RC = Int$$

The interpretation function has usual signature:
$\mathcal{E}_{ref} : Exp \rightarrow Fenv \rightarrow Env \rightarrow Sto \rightarrow (Val \times Sto)$
Plus something to decrement counts on the unused branches
$\mathcal{D}_{ref} : Exp \rightarrow Env \rightarrow Sto \times Sto$

`inc`: $Sto \rightarrow Loc \rightarrow Sto$
`dec`: $Sto \rightarrow Loc \rightarrow Sto$
`alloc`: $Sto \rightarrow (Loc \times Sto)$

Concrete Semantics

An example showing part of the concrete denotational semantics using reference counts are:

$$\begin{aligned}
\mathcal{E}_{ref}[\![\texttt{(if E1 E2 E3)} = \lambda f\ e\ s_0\ .\ & \\
& \textbf{let} < v1, s1 >= \mathcal{E}_{ref}[\![E1]\!] f\ e\ s_0 \\
& \textbf{in} \ldots \textit{type checking omitted} \ldots \\
& \ \textbf{if}\ v1 \\
& \quad \textbf{then}\ (\mathcal{E}_{ref}[\![E2]\!] f\ e\ (\mathcal{D}_{ref}\ [\![E3]\!] s1)) \\
& \quad \textbf{else}\ (\mathcal{E}_{ref}\ [\![E3]\!] f\ e(\ \mathcal{D}_{ref}\ [\![E2]\!] s1)) \\
& \quad \textbf{endif}\ \mathcal{E}_{ref}
\end{aligned}$$

The references to a vector in the unused branches are not counted.

Abstract Semantics

In the abstract semantics, assume every expression has a unique label. The abstract intpretation is very similar to the concrete one, but manipulates sets of pairs, rather than individual pairs.

$$\mathcal{E}_{abs} : \textit{Label} \rightarrow \textit{Fenv} \rightarrow \textit{Env} \rightarrow \textit{Sto} \rightarrow \mathcal{P}((\textit{Val} \times \textit{Sto}))$$

Store operation are on finite number of locations. Reference count operations are now finite and *sticky*: if you hit infinity you stay there.

Summary of Abstract Interpretation

- Idea is simple: program evaluation on an approximation to the value domain.
- An application of denotational semantics
- Several optimizations possible (mainly functional languages).
- Hard in general case.