CS4212 Cheatsheet AY23/24 Sem 1

by Richard Willie

x86 calling convention

- Callee save: %rbp, %rax, %r12-%r15
- Caller save: all others
- Parameters 1..6 go in: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Parameters 7+ go on the stack, n-th argument is located at $(((n-7)+2)\cdot 8)$(%rbp)
- $\mathrm{I}_{\mathrm{ret}}$ is located at 8(%rbp)
- Return value in %rax
- Caller protocol:
  1. Set up arguments
  2. Push $\mathrm{I}_{\mathrm{ret}}$, to know which instruction to execute when we return from the callee
  3. Set %rip to the instruction of the callee
  4. Push old %rbp
  5. Set %rbp to %rsp
  6. Allocate 128 bytes of scratch stack space

LL(1) parser

- Problem: We can't decide which S production to apply until we see the symbol after the first expression.
- Solution: Left-factor the grammar. There is a common S prefix for each choice, so add a non-terminal S' at the decision point. Also need to eliminate left-recursion.
- Construct predictive parsing table (non-terminal $\times$ input token $\rightarrow$ production).

LR(0) parser

- Parsing is a sequence of *shift* and *reduce* operations.
- Represent the parser automaton as a table of shape (state $\times$ (terminals + non-terminals)).
- Problems: shift/reduce conflict and reduce/reduce conflict.
- Such conflicts can often be resolved by using a look-ahead symbol: LR(1).
- Ambiguity in associativity/precedence usually lead to shift/reduce conflicts.

Values and substitution

- To substitute a (closed) value v for some variable x in an expression e:
  1. Replace all free occurences of x in e by v.
  2. In OCaml: written `subst v x e`
  3. In math: written $e\{v/x\}$

Free variables

- Free variables are defined in an outer scope.
- A term with no free variables is called *closed*.
- In math:

$$fv(\texttt{x}) = \{\texttt{x}\}$$
$$fv(\texttt{fun x -> exp}) = fv(\texttt{exp}) \cup \{x\}$$
$$fv(\texttt{exp}_1\texttt{exp}_2) = fv(\texttt{exp}_1) \cup fv(\texttt{exp}_2)$$

Substitution

- Function application is interpreted as *substitution*.
- If we naively substitute an open term, a bound variable might capture the free variable.

$$(\texttt{fun x -> (x y)})\{(\texttt{fun z -> x})/y\}$$
$$= \texttt{fun x -> (x (fun z -> x))}$$

Alpha equivalence

- The names of bound variables don't matter to the semantics.
- The names of free variables do matter.
- Two terms that differ only by consistent renaming of *bound* variables are called *alpha equivalent*.

Fixing substitution

- Consider: $e_1\{e_2/x\}$
- To avoid capture, we define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2.

$$(\texttt{fun x -> (x y)})\{(\texttt{fun z -> x})/y\}$$
$$= \texttt{fun x' -> (x' (fun z -> x))}$$

Semantic analysis

- The *semantic analysis* phase:
  1. Resolve symbol occurrences to declarations, e.g. undefined symbols.
  2. Type checking.
- Main data structure manipulated by semantic analysis: *symbol table*.

Scope checking

- Ocaml code:

```
let rec scope_check g exp =
  match exp with
  | Var x ->
    if member x g then ()
    else failwith (x ^ " not in scope")
  | App (exp1, exp2) ->
    ignore (scope_check bound exp1);
    scope_check bound exp2
  | Fun (x, exp) ->
    scope_check (union g x) exp
```

What is a type

- *Intrinsic view (Church-style)* – a type is syntactically part of a program.
  1. A program that cannot be typed is not a program.
  2. Types do not have inherent meaning – they are used to define the syntax.
- *Extrinsic view (Curry-style)* – a type is a property of a program.

What is a type system

- A type system consists of a system of judgements and inference rules.
- A *judgement* is a claim, which may or may not be valid, e.g. $G \vdash e : t$.
- An *Inference rule* is used to derive valid judgements from other valid judgements. It consists of a list of premises and one conclusion.

Type checking

- A *derivation* or *proof tree* has instances of judgements as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms*, i.e. rules with no premises.
- Goal of the type checker: verify that such tree exists.
- An *Inference rule* is used to derive valid judgements from other valid judgements. It consists of a list of premises and one conclusion.

Subtyping

- *Subtype relation*, e.g. Pos <: Int.
- Give rise to a *subtyping hierarchy*.
- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB), i.e. $\text{LUB}(T_1, T_2)$. It is also called the join operation, i.e. $T_1 \vee T_2$.
- A subtyping relation $T_1 <: T_2$ is *sound* if it approximates the underlying semantic subset relation.
- If $T_1 <: T_2$ implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 <: T_2$ is sound.
- Subsumption rule:

$$\frac{G \vdash e : T \qquad T <: S}{G \vdash e : S}$$

- Subtyping for function:

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{(T_1 \to T_2) <: (S_1 \to S_2)}$$

- Rationale: Need to convert an S_1 to T_1 and T_2 to S_2, so the argument type is *contravariant* and the output type is *covariant*.
- Width subtyping of immutable record:

$$\frac{m \leq n}{\{l_1 : T_1; \ldots; l_n : T_n\} <: \{l_1 : T_1; \ldots; l_m : T_m\}}$$

Subtyping and references

- Covariant reference types are unsound, e.g. should NonZero ref <: Int ref?
- Contravariant reference types are also unsound, i.e. if $T_1 <: T_2$ then ref T_2 <: ref T_1 is unsound.
- Conclusion: Mutable structures are invariant.

$$T_1 \text{ ref} <: T_2 \text{ ref implies } T_1 = T_2$$

Dataflow analysis

- Liveness:
  1. Domain: set of variables
  2. $\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
  3. $\text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$
- Reaching Definitions:
  1. Domain: set of nodes
  2. $\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']$
  3. $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$
- Available Expressions:
  1. Domain: set of nodes
  2. $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
  3. $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$
- The domain has structure that reflects the amount of information for each dataflow value.
- More information enables more optimizations, but may be unsafe, so we have to settle with less information.
- Dataflow values $\ell_1 \sqsubseteq \ell_2$ means ℓ_2 has at least as much information as ℓ_1, hence ℓ_2 is better for optimizations.
- For available expressions, larger sets of nodes are more informative, hence $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \subseteq \ell_2$.
- For liveness, smaller sets of variables are more informative, hence $\ell_1 \sqsubseteq \ell_2$ iff $\ell_1 \supseteq \ell_2$.

Register allocation

- Graph coloring algorithm:
  1. Build interference graph.
     (a) Pre-color nodes as necessary.
     (b) Add move related edges.
  2. Reduce the graph (building a stack of nodes to color).
     (a) Simplify the graph as much as possible without removing nodes that are move-related. Remaining nodes are high degree or move-related.
     (b) Coalesce move related notes using Brigg's or George's strategy.
     (c) Coalescing can reveal more nodes that can be simplified, so repeat 2(a) and 2(b) until no node can be simplified or coalesced.
     (d) If no nodes can be coalesced, remove/freeze a move-related edge and keep simplify/coalesce.
  3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
  4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
     (a) If a node must be spilled, insert spill code and return the whole algorithm from step 1.
- Conservating coalescing:
  1. *Brigg's strategy* – coalesce only when the resulting node has $< k$ neighbors with degree $\geq k$.
  2. *George's strategy* – coalesce x and y only when each neighbor of x is either a neighbor of y or has degree $< k$.