IA010: Principles of Programming Languages

Constraints

Achim Blumensath
blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Declarative programming

Describe **what** you want to compute, not **how**

(no side-effects, no state)

Advantages

- easier to reason about
- write separately and compose

Logic programming

write set of constraints and search for solution

Single-assignment variables

$$\langle expr \rangle ::= \ldots \mid \textbf{let}\ \langle id \rangle\ \textbf{;}\ \langle expr \rangle$$

```
let x;
let y;
x := 1;
x := 1; // ok
x := 2; // error
y := x+1;

let add(x,y,z) {
  z := x+y;
};
let u;
add(1,2,u);
```

```
let reverse(lst, ret) {
  let iter(lst, acc, ret) {
    case lst
    | []     => ret := acc
    | [x|xs] => iter(xs, [x|acc], ret)
  };

  iter(lst, [], ret)
};
```

```
let reverse(lst, ret) {
  let iter(lst, acc, ret) {
    case lst
    | []     => ret := acc
    | [x|xs] => iter(xs, [x|acc], ret)
  };

  iter(lst, [], ret)
};
```

```
reverse(L,R) :- iter (L, [], R).

iter([], A, A).
iter([H|T], A, R) :- iter(T, [H|A], R).
```

Unification

$$\langle expr \rangle ::= \ldots \mid \langle expr \rangle \texttt{ :=: } \langle expr \rangle$$

```
1 :=: x             x := 1
x :=: y             identifies x and y
[x,2] :=: [1,y]     x := 1 and y := 2
```

Unification algorithm

solve $u :=: v$

- If u is an uninitialised variable, set it to v.
- If v is an uninitialised variable, set it to u.
- If $u = m$ and $v = n$ are numbers, check that $m = n$.
- If $u = c(s_0, \ldots, s_{m-1})$ and $v = d(t_0, \ldots, t_{n-1})$ are constructors, check that $c = d$, $m = n$, and $s_i :=: t_i$, for all i.
- If $u = [l_0 = s_0, \ldots, l_{m-1} = s_{m-1}]$ and $v = [k_0 = t_0, \ldots, k_{n-1} = t_{n-1}]$ are records, find bijection $\varphi : m \to n$ such that $l_i = k_{\varphi(i)}$ and $s_i :=: t_{\varphi(i)}$, for all i.
- In all other cases, fail.

(In particular, we cannot unify function values.)

Unification algorithm

solve $u :=: v$

- If u is an uninitialised variable, set it to v.
- If v is an uninitialised variable, set it to u.
- If $u = m$ and $v = n$ are numbers, check that $m = n$.
- If $u = c(s_0, \ldots, s_{m-1})$ and $v = d(t_0, \ldots, t_{n-1})$ are constructors, check that $c = d$, $m = n$, and $s_i :=: t_i$, for all i.
- If $u = [l_0 = s_0, \ldots, l_{m-1} = s_{m-1}]$ and $v = [k_0 = t_0, \ldots, k_{n-1} = t_{n-1}]$ are records, find bijection $\varphi : m \to n$ such that $l_i = k_{\varphi(i)}$ and $s_i :=: t_{\varphi(i)}$, for all i.
- In all other cases, fail.

(In particular, we cannot unify function values.)

Notes

- two kinds of uninitialised values: unknown value, equal to other variable
- need to prevent infinite loops

Backtracking

$$\langle expr \rangle ::= \ldots \mid \textbf{choose} \,|\, \langle expr \rangle \ldots \,|\, \langle expr \rangle \mid \textbf{fail}$$

```
let is_one_or_two(x) {
  choose
  | x := 1
  | x := 2
};

is_one_or_two(1);  // ok
is_one_or_two(3);  // fail
```

Primitive operations

checkpoint k

- stores the current continuation and machine state

rewind

- fetches the continuation associated with the last checkpoint,
- restores the machine state to its previous state (deleting the last checkpoint),
- and calls the fetched continuation.

Primitive operations

checkpoint k

- stores the current continuation and machine state

rewind

- fetches the continuation associated with the last checkpoint,
- restores the machine state to its previous state (deleting the last checkpoint),
- and calls the fetched continuation.

```
choose | e1                    ⟹   e1
choose | e1 | e2 ... | en      ⟹   letcc k {
                                       checkpoint
                                         fun () {
                                           k(choose | e2 ... | en)
                                         };
                                       e1
                                     }
fail                           ⟹   rewind
```

Implementation

- store stack of checkpoints
- each checkpoint contains: continuation, list of modified variables
- `checkpoint k` puts `k` on the stack
- when we set a variable `x`, we add `x` to the top list
- `rewind` pops the stack, unsets all variables in the top list, and calls the stored continuation

Example

```
edge(a,b).
edge(b,c).
trans(X,Y) :- edge(X,Y).
trans(X,Y) :- edge(X,Z), trans(Z,Y).

let edge(x,y) {
  choose
  | { x := a; y := b; }
  | { x := b; y := c; }
}
let trans(x,y) {
  choose
  | edge(x,y)
  | { let z; edge(x,z); trans(z,y); }
}
```