Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

WS 2019/20
11. 11. 2019

Semantics of Programming Languages

Exercise Sheet 4

Exercise 4.1 Rule Inversion

Recall the evenness predicate *ev* from the lecture:

```
inductive ev :: "nat ⇒ bool" where
  ev0: "ev 0" |
  evSS: "ev n ⟹ ev (Suc (Suc n))"
```

Prove the converse of rule *evSS* using rule inversion. Hint: There are two ways to proceed. First, you can write a structured Isar-style proof using the *cases* method:

```
lemma "ev (Suc (Suc n)) ⟹ ev n"
proof −
  assume "ev (Suc (Suc n))" then show "ev n"
  proof (cases)

    ...

  qed
qed
```

Optional: Alternatively, you can write a more automated proof by using the **inductive_cases** command to generate elimination rules. These rules can then be used with "*auto elim*:". (If given the [*elim*] attribute, *auto* will use them by default.)

```
inductive_cases evSS_elim: "ev (Suc (Suc n))"
```

Next, prove that the natural number three (*Suc* (*Suc* (*Suc 0*))) is not even. Hint: You may proceed either with a structured proof, or with an automatic one. An automatic proof may require additional elimination rules from **inductive_cases**.

```
lemma "¬ ev (Suc (Suc (Suc 0)))"
```

Exercise 4.2 (Deterministic) labeled transition systems

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

A *labeled transition system* is a directed graph with edge labels. We represent it by a predicate that holds for the edges.

type_synonym $('q,'l)$ *lts* = "$'q \Rightarrow 'l \Rightarrow 'q \Rightarrow$ *bool*"

I.e., for an LTS δ over nodes of type $'q$ and labels of type $'l$, $\delta\ q\ l\ q'$ means that there is an edge from q to q' labeled with l.

A word from source node u to target node v is the sequence of edge labels one encounters when going from u to v.
Define a predicate *word*, such that *word* $\delta\ u\ w\ v$ holds iff w is a word from u to v.

inductive *word* :: "$('q,'l)$ *lts* $\Rightarrow 'q \Rightarrow 'l$ *list* $\Rightarrow 'q \Rightarrow$ *bool*" **for** δ

A *deterministic* LTS has at most one transition for each node and label

definition "*det* $\delta \equiv \forall\ q\ a\ q1\ q2.\ \delta\ q\ a\ q1 \wedge \delta\ q\ a\ q2 \longrightarrow q1 = q2$"

Show: For a deterministic LTS, the same word from the same source node leads to at most one target node, i.e., the target node is determined by the source node and the path

lemma
assumes *det*: "*det* δ"
shows "*word* $\delta\ q\ w\ q' \Longrightarrow$ *word* $\delta\ q\ w\ q'' \Longrightarrow q' = q''$"

Exercise 4.3 Counting Elements

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

Recall the count function, that counts how often a specified element occurs in a list:

fun *count* :: "$'a \Rightarrow 'a$ *list* $\Rightarrow$ *nat*" **where**
"*count* x [] = 0"
| "*count* x $(y\#ys)$ = (*if* $x{=}y$ *then Suc* (*count* x *ys*) *else count* x *ys*)"

Show that, if an element occurs in the list (its count is positive), the list can be split into a prefix not containing the element, the element itself, and a suffix containing the element one times less

lemma "*count* x *xs* = *Suc* $n \Longrightarrow \exists\, p\ s.\ xs = p\ @\ x\ \#\ s \wedge$ *count* $x\ p = 0 \wedge$ *count* $x\ s = n$"
theory *Homework*
imports *Main* "*HOL−IMP.AExp*"
begin

Homework 4.1 Cycles in Graphs

Submission until Monday, November 18, 2019, 10:00am.

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

Similarly to the labeled transition systems from the tutorial, we can view directed graphs simply as an edge relation $E :: ('a \Rightarrow 'a \Rightarrow bool)$. Your first task is to the define notion

of path in a graph inductively as a predicate *path E p*, where p contains the nodes of the path. A path is either a trivial path $[v]$ from a node v to itself or of the form $[v_1, v_2, \ldots, v_n]$ such that $E\ v_1\ v_2, \ldots, E\ v_{n-1}\ v_n$ all hold.

inductive *path* :: *"('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool"* **for** *E* **where**

We now want prove a sufficient condition that guarantees that a given graph does *not* have a cycle through a certain node v. The idea is to give a numbering $f :: 'a \Rightarrow nat$ such that

- for all edges $E\ a\ b$, the value of f is non-decreasing
- for any edge $E\ v\ w$ out of v, the value of f is increasing

Prove that under these conditions, there is no cycle trough v:

theorem *no_cycle*:
fixes *f* :: *"'a ⇒ nat"*
assumes *"∀ a b. E a b ⟶ f a ≤ f b"* *"∀ w. E v w ⟶ f v < f w"*
shows *"¬ (∃ xs. path E (v # xs @ [v]))"*

Homework 4.2 Sharing Sub-Expressions

Submission until Monday, November 18, 2019, 10:00am.

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

We extend the arithmetic expressions known from the lecture with a construct for *Let*-expressions:

datatype *lexp = N int | V vname | Plus lexp lexp | Let vname lexp lexp*

An expression *Let x e a* binds the expression e to variable x in a:

fun *lval* :: *"lexp ⇒ state ⇒ val"* **where**
"lval (N n) s = n" |
"lval (V x) s = s x" |
"lval (Plus a₁ a₂) s = lval a₁ s + lval a₂ s" |
"lval (Let x a b) s = lval b (s(x := lval a s))"

lemma *example*:
"lval (Let ''x'' (N 5) (Let ''y'' (V ''x'') (Plus (V ''x'') (Plus (V ''y'') (V ''x''))))) <> = 15"
by *eval*

Let-expressions allow us to share subexpressions to avoid recomputation. The expression $((a + 3) + 4) + ((a + 3) + 4)$ could be re-written as *Let* $x = a + 3 + 4$ *in* $x + x$, for instance. The goal of this exercise is to define a function *linearize* that performs such rewriting automatically. Our function will assume that the given expression does not

contain any *Let*-expressions and will only replace occurrences of repeated expressions of the form $a + b$, i.e. no variables or constants.

Step 1 We define a function *replace* that performs a single replacement of a subterm by a variable, i.e. *replace e x a* replaces every occurrence of *e* by *V x* in *a*. Fill in the case for *Plus a b*:

```
fun replace :: "lexp ⇒ vname ⇒ lexp ⇒ lexp" where
"replace e x (Let u a b) = Let u (replace e x a) (replace e x b)"| "replace e x a = a"
```

Step 2 We define the auxiliary functions *vars_of* and *vars_of* that collect the whole set of variables, and the set of variables that are bound by a *Let*-expression in a given expression, respectively:

```
fun vars_of :: "lexp ⇒ string set" where
  "vars_of (N _) = {}"
| "vars_of (V x) = {x}"
| "vars_of (Plus a b) = vars_of a ∪ vars_of b"
| "vars_of (Let x a b) = {x} ∪ vars_of a ∪ vars_of b"

fun bounds_of :: "lexp ⇒ string set" where
  "bounds_of (N _) = {}"
| "bounds_of (V x) = {}"
| "bounds_of (Plus a b) = bounds_of a ∪ bounds_of b"
| "bounds_of (Let x a b) = {x} ∪ bounds_of a ∪ bounds_of b"
```

Show that updating the state for a variable x does not occur in a does not influence the result of evaluating a:

```
theorem lval_upd_state_same:
  "x ∉ vars_of a ⟹ lval a (s(x := v)) = lval a s"
```

Step 3 Show that evaluating a on s yields the same result as first replacing x by e in a, and then evaluating the result on $s(x := lval\ e\ s)$:

```
theorem lval_replace:
  assumes "x ∉ vars_of a" "bounds_of a ∩ vars_of e = {}"
  shows "lval (replace e x a) (s(x := lval e s)) = lval a s"
```

Step 4 We are now ready to define the function *linearize*. We first define a function *collect* that collects all-subexpressions of a given arithmetic expression

```
fun collect :: "lexp ⇒ lexp list" where
  "collect (N n) = []"
| "collect (V _) = []"
| "collect (Plus a b) = collect a @ Plus a b # collect b"
| "collect (Let x a b) = collect a @ collect b"
```

and provide the following helper functions:

```
fun invent_names :: "nat ⇒ string list" where
  "invent_names 0 = []"
| "invent_names (Suc n) = replicate (Suc n) (CHR ''v'') # invent_names n"

fun duplicates :: "'a list ⇒ 'a list" where
  "duplicates [] = []"
| "duplicates (x # xs) = (if x ∈ set xs then x # duplicates xs else duplicates xs)"
```

Define the function *linearize* using the following template:

```
definition linearize' :: "lexp ⇒ lexp" where
  "linearize' e = (let
    exps = undefined;
    names = undefined;
    m = zip exps names
  in fold (λ(a, x) e. Let x a (replace a x e)) m e)"
```

The function should only use names of the form v, vv, vvv, ... to bind expressions to variables. Your functions should remove all occurrences of duplicate expressions. In addition, it should pass the following test cases:

```
theorem test_case1:
  "linearize (Plus (Plus (Plus (V ''a'') (N 3)) (N 4)) (Plus (V ''a'') (N 3)))
= Let ''v'' (Plus (V ''a'') (N 3)) (Plus (Plus (V ''v'') (N 4)) (V ''v''))"
theorem test_case2:
  "linearize (Plus (Plus (Plus (V ''a'') (N 3)) (N 4)) (Plus (Plus (V ''a'') (N 3)) (N 4)))
= Let ''v'' (Plus (V ''a'') (N 3)) (Let ''vv'' (Plus (V ''v'') (N 4)) (Plus (V ''vv'') (V ''vv'')))"
```

(Bonus) Step 6 As a bonus exercise, you can try to prove correctness of *linearize*:

```
lemma linearize_correct:
  assumes "∀x. x ∈ vars_of e ⟶ CHR ''v'' ∉ set x" "bounds_of e = {}"
  shows "lval (linearize e) s = lval e s"
```

We will reward up to 3 bonus points. Partial credit for incomplete attempts may be given. Bonus points count "on your side": Bonus points are added to your final homework score but do not count towards the maximum number of homework points that can be achieved.