Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dipl.-Inf. Achim D. Brucker
Dr. Burkhart Wolff

Computer-supported Modeling and Reasoning

http://www.infsec.ethz.ch/education/permanent/csmr/

(rev. 16814)

Submission date: –

HOL: Well-founded and Primitive Recursion

In this exercise, we will deepen our knowledge on well-founded orderings and induction as well as its applications in form of recursive definitions.

1 Recursive Definitions

1.1 Primitive recursion

Isabelle provides a syntactic front-end for defining an important subclass of well-founded recursions, namely *primitive recursive* functions, e.g.:

primitive recursive
primrec

```
primrec
  add_0:    "0 + n = n"
  add_Suc:  "Suc m + n = Suc (m + n)"

primrec
  diff_0 :   "m - 0 = m"
  diff_Suc : "m - Suc n = (case m - n of
                             0   => 0
                           | Suc k => k)"
```

The general form of a primitive recursive definitions in Isabelle is:

primrec
  $name_1$: "*rule*"
  ⋮
  $name_n$: "*rule*"

reduction rules where *rule* are *reduction rules* (as usual, the names $name_1 \ldots name_n$ are optional). The reduction rules specify one or more equations of the form

$$f\ x_1\ \ldots\ x_n(C\ y_1\ \ldots\ y_n)\ z_1\ \ldots\ z_n = r$$

such that C is a constructor of the datatype (e.g. Suc in our first example), r contains only free variables on the left-hand side, and all recursive calls in r are of the form $f\ \ldots\ y_i\ \ldots$ for some i.

1.2 General Recursive Definitions

recdef Isabelle also offers a way for declaring functions using general well-founded recursion: **recdef**. Using **recdef**, you can declare functions involving nested recursion and pattern-matching, e.g. we can define the Fibonacci function:

```
consts fib :: "nat ⇒ nat"
recdef fib "less_than"
      "fib 0 = 0"
      "fib 1 = 1"
      "fib (Suc(Suc x)) = (fib x + fib (Suc x))"
```

where les_than is the "less than" on the natural numbers.

The general form of a recursive definitions in Isabelle is:

primrec *function rule*
    congs   "*rules*"
    simpset "*rules*"
  $name_1$: "*rule*"
  ⋮
  $name_n$: "*rule*"

where *function* is the functions name and *rule* a HOL expression for the well-founded termination relation (Isabelle provides several built-in relations such as less_than or measure). With the to *optional* arguments congs and simpset one can influence the set of congurences rules and the simpset used during the termination proof. Finally, the *rules* are specifing the "computational" recursive equations.

2 Exercises

2.1 Exercise 40

Prove the following consequences of well-founded orderings:

1. a well-founded ordering is not symmetric:

   lemma wf_not_sym: "wf(r) $\Longrightarrow \forall$ a x. (a,x)$\in$r $\longrightarrow$ (x,a)$\notin$r"

2. a well-founded ordering contains minimal elements:

   lemma wf_minimal: "wf r $\Longrightarrow \exists$ x. $\forall$ y. (y,x) $\notin$ r^+"

3. a subrelation of a well-founded ordering is well-founded:

   lemma wf_subrel: "wf(p) $\Longrightarrow \forall$ r. r $\subseteq$ p $\longrightarrow$ ($\exists$ x. $\forall$ y. (y,x) $\notin$ r^+)"

4. a well-founded ordering satisfies characterization (1):

   lemma wf_eq_minimal2:
   "wf(p) = ($\forall$ r. (r$\neq${} $\wedge$r $\subseteq$ p) $\longrightarrow$ ($\exists$ x $\in$ Domain r. ($\forall$ y. (y,x) $\notin$ r)))"

Hint: Look up the various theorems about wellfounded orderings that Isabelle provides (wf_induct, wf_empty, wf_subset, wf_not_sym, wf_not_refl, wf_trancl, wf_acyclic, and wfrec_def) and use them as you like.

2.2 Exercise 41

1. Define a the recursor iter f n in terms of the well-founded recursor wfrec and the theory of the natural numbers. Derive from your definition the properties:

   lemma iter_0 : " iter 0 g = (λ x. x)"
   lemma iter_Suc : " iter (Suc n) g = g $\circ$ (iter n g)"

2. Define the addition add, the multiplication mult, the exponentiation exp and the sumup operation sumup (sumup 3 = 1 + 2 + 3) on natural numbers.

   Use in at least two definitions the iter-recursor and derive the usual computational equations; in the other cases, you may use a **primrec** construct.

2.3 Exercise 42 — "The approximation theorem of **lfp**"

In lecture "HOL: Fixpoints" we have seen the theorem:

$$(\forall S.\ f(\bigcup S) = \bigcup(f \text{ ` } S)) \Longrightarrow \bigcup_{n \in N} f^n(\emptyset) = \mathit{lfp}\ f$$

i.e. under a certain condition, a fix-point can be seen as a limit of an approximation process. This condition is also called *continuity of f*. Under an obvious alternative constraint, namely that the fix-point must be reachable after finitely many steps, this principle is of practical importance, for example in data-flow analysis algorithms (such as the Java Byte-code Verifier).

Prove one of the following versions of the approximation theorem:

1. **lemma** lfp_approximable_if_finite :
   ⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
   ⟹ (UN n:UNIV. (iter n f {})) = lfp f

2. **lemma** lfp_approximable_if_cont :
   ⟦(⋀S. f (Union S) = Union (f ‘ S))⟧
   ⟹ (UN n:UNIV. (iter n f {})) = lfp f

For the first option, we suggest the following intermediate lemmas:

1. mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f
2. ⟦mono f; ∃ m. f (iter m f {}) = (iter m f {})⟧
   ⟹ lfp f ≤(UN n:UNIV. (iter n f {}))

For the second option, we suggest the following milestones:

1. mono f ⟹(UN n:UNIV . (iter n f {})) ≤ lfp f
2. (∀ S. f (Union S) = Union (f ‘ S)) ⟹ mono f
3. (UN n:UNIV. iter (Suc n) f {}) = (UN n:{m. 0 < m}. (iter n f {}))
4. (UN n:UNIV. g (n::nat)) = (UN n:{m. 0 < m}. (g n))Un (g 0)
5. (∀ S. f (⋃S)= ⋃f ‘S)
   ⟹ f ($\bigcup_n$iter n f {}) = ($\bigcup_n$iter n f {})
6. (∀ S. f (Union S) = Union (f ‘ S))
   ⟹ f (UN n:UNIV. (iter n f {})) = (UN n:UNIV. f (iter n f {}))

7. $\bigwedge$S. f (Union S) = Union (f ` S)) $\Longrightarrow$ lfp f $\leq$ (UN n:UNIV. (iter n f {}))

Hint: Look up the various theorems about the inclusion operation that Isabelle provides (rev_subsetD, lfp_unfold , monoD, Un_upper1, Un_absorb1, image_Collect) and use them as you like.