Types Summer School 2007
Why/Caduceus-lab

Jean-Christophe Filliâtre

Here are some exercises in program verification, with Why and Caduceus. All exercises are independent. Feel free to do them in the order of your choice.

1 Installing the software

The three tools are available on the live-CD, together with the three automatic provers Simplify, Ergo and Yices.

In case you are not using the live-CD, you can download Why/Caduceus from `http://why.lri.fr/` (a single tarball for both, available as source and Linux binaries). Linux binaries for Simplify, Ergo and Yices are available at `http://www.lri.fr/~filliatr/types-summer-school-2007/provers/`. Why and Caduceus manuals are available on-line at `http://why.lri.fr/`.

2 Why

The simplest way to use Why is to build a source file `f.why` and then to run the graphical user interface with `gwhy f.why`. Then each automatic prover can be run on proof obligations by clicking on its name. A given proof obligation can be displayed by selecting it in the list (on the left side). The `gwhy` command line option `-split-user-conj` can be specified to split proof obligations into several pieces, as soon as they are built with conjunctions. To use an interactive prover such as Coq, run `why --coq f.why` and then edit `f_why.v` with CoqIDE.

2.1 McCarthy's 91 function

McCarthy's 91 function is the function f from $\mathbb{Z}$ to $\mathbb{Z}$ defined by

$$\left\{ \begin{array}{lcll} f(n) & = & f(f(n+11)) & \text{if } n \leq 100 \\ f(n) & = & n-10 & \text{otherwise.} \end{array} \right.$$

1. Define function f in Why. The Why syntax for a recursive function is

    ```
    let rec f (n:int) : int = ...
    ```

2. Annotate f in order to prove that $f(n)$ is 91 when $n \leq 100$ and $n-10$ otherwise.

3. Prove the termination of f by inserting the following variant

    ```
    let rec f (n:int) : int { variant max(0,101-n) } = ...
    ```

    Since `max` is not a primitive function, you must introduce it with a `logic` and axiomatize it with an `axiom`.

2.2 Fibonacci Function

1. Introduce the Fibonacci function F with a logic and three axioms. We recall that $F(0) = F(1) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.

2. Define a recursive function f_1 computing F (with a naive, *i.e.* exponential, algorithm). Prove its correctness and termination.

3. Define a function f_2 computing F using a linear algorithm which maintains $F(n-1)$ and $F(n)$ in two references. Prove its correctness and termination.

4. Define a third function f_3 computing $F(n)$, using the same linear algorithm but using a recursive function instead of a loop. Note how the loop invariant is naturally transformed a precondition.

3 Caduceus

Given an annotated C source file `f.c`, Caduceus is simply invoked with the command `caduceus f.c`. It creates an input file for `make` in `f.makefile`. Then the user interface `gwhy` can be launched with

```
make -f f.makefile gui
```

To split proof obligations, use Caduceus option `-why-opt -split-user-conj`. To enable separation analysis, use Caduceus option `-separation`.

3.1 Basics of Loop Annotations

Let us consider a simple loop iterating over all integers from 0 to $n-1$, assuming that $n \geq 0$. It can be written as follows:

```
for (i = 0; i < n; i++) ...
```

In order to maintain some information about `i` inside the loop, we need at least a loop invariant such as:

```
//@ invariant 0 <= i <= n
for (i = 0; i < n; i++) ...
```

Note that we need to write `i <= n` and not `i < n` since the invariant must also hold at the end of the last execution of the loop body, when `i = n`. In order to prove the termination of such a loop, we need to introduce a variant, such as:

```
//@ invariant ... variant n-i
for (i = 0; i < n; i++) ...
```

Here are some basic exercices related to loop invariants and variants:

1. Prove the termination of the following functions:

```
void loop1(int n) { while (n > 0) n--; }
void loop2(int n) { while (n < 100) n++; }
```

2. Prove the correctness and termination of this program:

```
//@ ensures \result == 0
int loop3() {
  int i = 100;
  while (i > 0) i--;
  return i;
}
```

3.2 All Zeros

The purpose of this exercise is to define a function to check, given an array t and an integer n, whether the elements $t[0], \ldots, t[n-1]$ are all zeros or not. Thus the specification of the function will be the following:

```
/*@ requires n >= 0 && \valid_range(t,0,n)
    ensures  \result <=> \forall int i; 0<=i<n => t[i]==0 */
int all_zeros(int t[], int n) { ... }
```

1. Prove the correctness and termination of this first implementation:

```
int all_zeros_0(int t[], int n) {
  int k;
  /*@ invariant ... variant ... */
  for (k = 0; k < n; k++) if (t[k]) return 0;
  return 1;
}
```

2. Same question for this second implementation:

```
int all_zeros(int t[], int n) {
  /*@ invariant ... variant ... */
  while (--n>=0 && !t[n]);
  return n < 0;
}
```

3. Is this third implementation correct? If so, prove it; if not, find a counterexample.

```
int all_zeros(int t[], int n) {
  int k = 0;
  while (k < n && !t[k++]);
  return k == n;
}
```

3.3 Pointer Arithmetic

Prove the absence of runtime error in the following C code:

```
//@ requires size >= 0 && \valid_range(p,0,size-1)
void erase(int *p, int size){
  while (size--) *p++ = 0;
}
```

3.4 Linked Lists

In this exercise we consider linked lists of integers, defined as follows:

```
typedef struct struct_list {
  int hd;
  struct struct_list * tl;
} *list;
```

The empty list is simply represented by a null pointer:

```
#define NULL ((void*)0)
```

We want to verify the following function which searches the list `p` for the value `v`:

```
list search(list p, int v) {
  while (p != NULL && p->hd != v) p = p->tl;
  return p;
}
```

It returns either a pointer to a cell containing `v` or `NULL` if the search is not successful. Obviously, this program breaks if the list `p` is not well-formed (*i.e.* contains invalid pointers) or loops forever if the list is cyclic and does not contain `v`. Thus we need to characterize well-formed finite lists. For this purpose, we introduce the following predicate:

```
/*@ predicate is_list(list p) reads p->tl */
```

Up to now, this is an uninterpreted predicate. The declaration `reads p->tl` indicates that the meaning of `is_list` may depend on `p->tl` (to allow Caduceus to interpret `is_list` as a Why predicate with the corresponding part of the model as argument).

1. Complete the following axiom for `is_list`:

   ```
   /*@ axiom is_list_def : \forall list p; is_list(p) <=> (...) */
   ```

2. Prove the correctness of function `search`.

3. Prove the termination of function `search`. For this purpose, introduce the length of a linked list as a logical function:

   ```
   /*@ logic int length(list p) reads p->tl */
   ```

   and add the necessary axioms for `length`.