Functional Programming 1 — 1DL330

Variants

Tjark Weber

1 Motivation

Whenever you write a function, you should convince yourself—and sometimes you will need to convince other people too—that your implementation is correct. One aspect of correctness is that your function should *terminate* for all valid inputs, i.e., for all inputs that meet the function's precondition.

Certainly, it is easy to write recursive functions that do not terminate.[1] So how do you know that a recursive function terminates? For instance, how do you know that the following function terminates for all valid inputs?

```
(* f n
   TYPE: int -> int
   PRE: n >= 0
   POST: 2n
 *)
fun f 0 = 0
  | f n = 2 + f (n-1)
```

Well, you can use induction to prove that this function terminates for all valid inputs.[2] But induction proofs often become tedious, especially when the function in question is more complicated.

Variants are a mathematical tool: they allow you to prove more conveniently that (recursive) functions terminate.

2 Definition and Theorems

A *variant* for a (recursive) function is any expression over the function's arguments that takes values in some set A such that

- A is (totally) ordered; moreover, there are no infinite descending chains $v_0 > v_1 > \ldots$ in A; and
- for any recursive call, the variant decreases strictly.

Some explanations are in order. First, variants are useful because of the following theorem (whose proof we will omit).

[1] **Exercise:** Give an example of a recursive function that does not terminate for all valid inputs.

[2] **Exercise:** Do this now! Use induction to prove that `f n` terminates for all integers $\mathsf{n} \geq 0$.

Theorem. Every function that has a variant terminates.

Therefore, if you want to prove that a function terminates, it is enough to find a variant for the function. Instead of proving termination explicitly, e.g., by induction, you can then rely on the above theorem.

The converse is true, as well.

Theorem. Every function that terminates has a variant.

In other words, variants exist for all terminating functions. If you understand variants, you'll never have to use induction again to prove termination! (Of course, you may still need to use induction for other purposes.)

3 Examples

When you want to find a variant for a (recursive) function, you have to choose two things: first, the set A in which the variant will take values, and second, the actual variant, i.e., an expression over the function's arguments that is strictly decreasing for any recursive call.

Let's first look at the requirements for A. According to our definition of variants, A must be an ordered set, and it must not contain any infinite descending chains. In practice, a common choice for A is the set of non-negative integers $\mathbb{N} := \{0, 1, 2, \ldots\}$, with their usual order: $0 < 1 < 2 < \ldots$. Clearly, there are no infinite descending chains in this set: if you start at some number n, you can descend at most n times. You can also consider other choices for A: e.g., the set $\mathbb{N} \times \mathbb{N}$ of pairs of non-negative integers, ordered lexicographically, or some set of ordinals with their usual order. Again, neither of these sets contains any infinite descending chains.

On the other hand, the set of integers $\{\ldots, -1, 0, 1, \ldots\}$ with their usual order would *not* be a valid choice for A, as this set contains an infinite descending chain: $1 > 0 > -1 > \ldots$. Also the set of non-negative real numbers $\{x \in \mathbb{R} \mid x \geq 0\}$ with their usual order would not be a valid choice for A: this set likewise contains an infinite descending chain.[3]

Having chosen the set A—once again, in practice $A = \mathbb{N}$ often works just fine—you still need to find the actual variant for the (recursive) function whose termination you want to prove. According to our definition, the variant is an expression over the function's arguments that takes values in A, and that decreases strictly for any recursive call.

3.1 Example: f

Suppose we have chosen $A = \mathbb{N}$. Let us look at some expressions, and discuss whether they are variants for the function `f` from Section 1:

[3] **Exercise:** Find an infinite descending chain in the set of non-negative real numbers.

Expression	Variant	Explanation
n	yes	$n \geq 0$ implies $n \in \mathbb{N}$, and for the recursive call we have: $n \geq 0$ and $n \neq 0$ implies $n - 1 \in \mathbb{N}$, and $n > n - 1$.
True	no	This expression does not take values in $\mathbb{N}$.
x	no	This is not an expression over f's arguments. What is the value of x?
$n - 1$	no	For $n = 0$, the value of this expression is not in $\mathbb{N}$. (However, for a different choice of A, this could indeed be a variant.)
42	no	This expression is not strictly decreasing for every recursive call of f: replacing n by $n - 1$ in 42 yields 42 again.
n^2	yes	$n \geq 0$ implies $n^2 \in \mathbb{N}$, and for the recursive call we have: $n \geq 0$ and $n \neq 0$ implies $(n-1)^2 \in \mathbb{N}$, and $n^2 > (n-1)^2$.

3.2 Example: rev

For a slightly more advanced example, consider the following implementation of list reversal:

```
(* rev acc xs
   TYPE: 'a list -> 'a list -> 'a list
   PRE: True
   POST: (xs reversed) @ acc
   EXAMPLE: rev [3] [1,2] = [2,1,3]
*)
fun rev acc [] = acc
  | rev acc (y::ys) = rev (y::acc) ys
```

Intuitively, the function rev terminates because its second argument, xs, gets shorter with every recursive call. We can make this intuition formal by using length xs as a variant for rev. Clearly,

- length xs is an expression over acc and xs, the arguments of rev. (Note that rev's first argument, acc, actually gets longer with every recursive call. Therefore, acc is not helpful to prove termination of rev, and is consequently not mentioned in our variant.)
- length xs takes values in the set $\mathbb{N}$ of non-negative integers. This is a totally ordered set that has no infinite descending chains. (So we have again chosen $A = \mathbb{N}$.)
- There is one recursive call in the implementation of rev, and this call is made only when the second argument is a non-empty list, i.e., when xs is of the form y::ys for some y and ys. In this case, the value of the variant for the top-level call is length xs = length (y::ys) = length ys + 1, and the value of the variant for the recursive call is length ys. (Remember that our variant is simply the length of the function's second argument.) Since length ys + 1 > length ys, the variant is strictly decreasing for the recursive call.

Therefore, length xs satisfies all requirements on a variant.

3.3 Example: acker

Consider an implementation of the Ackermann function:

```
(* acker m n
   TYPE: int -> int -> int
   PRE: m>=0, n>=0
   POST: A(m,n), where A is the Ackermann-Peter function
   EXAMPLE: A(4,2) is an integer of 19,729 decimal digits
 *)
fun acker 0 n = n+1
  | acker m 0 = acker (m-1) 1
  | acker m n = acker (m-1) (acker m (n-1))
```

The Ackermann function is famous in computability theory: it is an example of a total function that is computable but not primitive-recursive. Here, however, we are merely concerned with its termination.

Finding a variant that takes non-negative integer values is not easy for this function. Instead, we choose $A = \mathbb{N} \times \mathbb{N}$, i.e., the set of pairs of non-negative integers, ordered lexicographically: $(a, b) <_{\text{lex}} (c, d)$ iff $a < c$ or ($a = c$ and $b < d$), where $<$ is the usual order on non-negative integers.[4] Now, consider the expression (m, n). Clearly,

- (m, n) is an expression over m and n, the arguments of acker.
- For valid arguments, i.e., $\mathsf{m} \geq 0$ and $\mathsf{n} \geq 0$, (m, n) takes values in $A = \mathbb{N} \times \mathbb{N}$.
- There are three recursive calls in the implementation of acker. For each recursive call, we have to check that (m, n) decreases strictly.

  1. When $\mathsf{m} = 0$, the first clause of acker applies. Therefore, the recursive call in the second clause of acker is made only when $\mathsf{m} > 0$. In this case, $(\mathsf{m}-1, 1) \in A$, and $(\mathsf{m}, 0) >_{\text{lex}} (\mathsf{m} - 1, 1)$.
  2. The third clause only applies when $\mathsf{m} > 0$ and $\mathsf{n} > 0$. In this case, for the inner (i.e., nested) recursive call we have $(\mathsf{m}, \mathsf{n} - 1) \in A$, and $(\mathsf{m}, \mathsf{n}) >_{\text{lex}} (\mathsf{m}, \mathsf{n} - 1)$.
  3. For the outermost recursive call in the third clause, we will need the fact that acker, if it terminates, returns a non-negative integer value. (This can be proved by induction.) We then have $(\mathsf{m} - 1, \mathsf{acker}\ \mathsf{m}\ (\mathsf{n} - 1)) \in A$, and $(\mathsf{m}, \mathsf{n}) >_{\text{lex}} (\mathsf{m} - 1, \mathsf{acker}\ \mathsf{m}\ (\mathsf{n} - 1))$.

Therefore, (m, n) is a variant for acker.

4 Concluding Remarks

As you can see from the examples in Section 3.1, there may be more than one variant for a given function. (In fact, any terminating function has infinitely many variants.) It doesn't matter which of these variants you choose—they all prove termination. Of course, simpler variants should be preferred over more complicated ones.

We have phrased the definition of variants so that it can be applied also to non-recursive functions. However, non-recursive functions always terminate, provided they only call other functions that terminate. (Note that there are no loops—and hence no infinite loops—in functional programming.) Therefore, variants for non-recursive functions

[4] **Exercise:** Prove that this defines a total order on $\mathbb{N} \times \mathbb{N}$, and that there are no infinite descending chains. What is the smallest element of $\mathbb{N} \times \mathbb{N}$ with respect to the lexicographic order?

are not very interesting. In particular, the requirement that the variant must strictly decrease for any recursive call is trivially satisfied. The following is a (trivial) variant for any non-recursive function: $\bullet$, where $A = \{\bullet\}$.[5]

Note that variants are *not* specific to functional programming. Variants are a tool to prove termination, and termination is also an issue for imperative or object-oriented programs. In imperative programs, non-termination is often caused by infinite loops, and the variants that we have considered here are closely related to loop variants in imperative programming.

See the lecture slides and also this question on Stack Overflow for additional explanations and further examples.

[5] **Exercise:** Check our definition of variants to confirm that this is indeed a variant for any non-recursive function.