# Defining Recursive Functions in Isabelle/HOL

Alexander Krauss

**Abstract**

This tutorial describes the use of the new *function* package, which provides general recursive function definitions for Isabelle/HOL. We start with very simple examples and then gradually move on to more advanced topics such as manual termination proofs, nested recursion, partiality, tail recursion and congruence rules.

## 1 Introduction

Starting from Isabelle 2007, new facilities for recursive function definitions [2] are available. They provide better support for general recursive definitions than previous packages. But despite all tool support, function definitions can sometimes be a difficult thing.

This tutorial is an example-guided introduction to the practical use of the package and related tools. It should help you get started with defining functions quickly. For the more difficult definitions we will discuss what problems can arise, and how they can be solved.

We assume that you have mastered the fundamentals of Isabelle/HOL and are able to write basic specifications and proofs. To start out with Isabelle in general, consult the Isabelle/HOL tutorial [4].

**Structure of this tutorial.** Section 2 introduces the syntax and basic operation of the **fun** command, which provides full automation with reasonable default behavior. The impatient reader can stop after that section, and consult the remaining sections only when needed. Section 3 introduces the more verbose **function** command which gives fine-grained control. This form should be used whenever the short form fails. After that we discuss more specialized issues: termination, mutual, nested and higher-order recursion, partiality, pattern matching and others.

**Some background.** Following the LCF tradition, the package is realized as a definitional extension: Recursive definitions are internally transformed into a non-recursive form, such that the function can be defined using standard definition facilities. Then the recursive specification is derived from the primitive definition. This is a complex task, but it is fully automated and mostly transparent to the user. Definitional extensions are valuable because they are conservative by construction: The "new" concept of general wellfounded recursion is completely reduced to existing principles.

The new **function** command, and its short form **fun** have mostly replaced the traditional **recdef** command [5]. They solve a few of technical issues around **recdef**, and allow definitions which were not previously possible.

# 2 Function Definitions for Dummies

In most cases, defining a recursive function is just as simple as other definitions:

```
fun fib :: "nat ⇒ nat"
where
  "fib 0 = 1"
| "fib (Suc 0) = 1"
| "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

The syntax is rather self-explanatory: We introduce a function by giving its name, its type, and a set of defining recursive equations. If we leave out the type, the most general type will be inferred, which can sometimes lead to surprises: Since both 1 and + are overloaded, we would end up with `fib :: nat ⇒ 'a::{one,plus}`.

The function always terminates, since its argument gets smaller in every recursive call. Since HOL is a logic of total functions, termination is a fundamental requirement to prevent inconsistencies[1]. Isabelle tries to prove termination automatically when a definition is made. In §4, we will look at cases where this fails and see what to do then.

## 2.1 Pattern matching

Like in functional programming, we can use pattern matching to define functions. At the moment we will only consider *constructor patterns*, which only consist of datatype constructors and variables. Furthermore, patterns must be linear, i.e. all variables on the left hand side of an equation must be distinct. In §6 we discuss more general pattern matching.

If patterns overlap, the order of the equations is taken into account. The following function inserts a fixed element between any two elements of a list:

```
fun sep :: "'a ⇒ 'a list ⇒ 'a list"
where
  "sep a (x#y#xs) = x # a # sep a (y # xs)"
| "sep a xs       = xs"
```

Overlapping patterns are interpreted as "increments" to what is already there: The second equation is only meant for the cases where the first one does not match. Consequently, Isabelle replaces it internally by the remaining cases, making the patterns disjoint:

```
thm sep.simps
```

```
sep a (x # y # xs) = x # a # sep a (y # xs)
sep a [] = []
sep a [v] = [v]
```

[1] From the "definition" `f(n) = f(n) + 1` we could prove `0 = 1` by subtracting `f(n)` on both sides.

The equations from function definitions are automatically used in simplification:

```
lemma "sep 0 [1, 2, 3] = [1, 0, 2, 0, 3]"
by simp
```

## 2.2 Induction

Isabelle provides customized induction rules for recursive functions. These rules follow the recursive structure of the definition. Here is the rule `sep.induct` arising from the above definition of `sep`:

```
⟦⋀a x y xs. ?P a (y # xs) ⟹ ?P a (x # y # xs); ⋀a. ?P a []; ⋀a v. ?P
a [v]⟧
⟹ ?P ?a0.0 ?a1.0
```

We have a step case for list with at least two elements, and two base cases for the zero- and the one-element list. Here is a simple proof about `sep` and `map`

```
lemma "map f (sep x ys) = sep (f x) (map f ys)"
apply (induct x ys rule: sep.induct)
```

We get three cases, like in the definition.

```
1. ⋀a x y xs.
      map f (sep a (y # xs)) = sep (f a) (map f (y # xs)) ⟹
      map f (sep a (x # y # xs)) = sep (f a) (map f (x # y # xs))
2. ⋀a. map f (sep a []) = sep (f a) (map f [])
3. ⋀a v. map f (sep a [v]) = sep (f a) (map f [v])
```

```
apply auto
done
```

With the **fun** command, you can define about 80% of the functions that occur in practice. The rest of this tutorial explains the remaining 20%.

# 3 fun vs. function

The **fun** command provides a convenient shorthand notation for simple function definitions. In this mode, Isabelle tries to solve all the necessary proof obligations automatically. If any proof fails, the definition is rejected. This can either mean that the definition is indeed faulty, or that the default proof procedures are just not smart enough (or rather: not designed) to handle the definition.

By expanding the abbreviation to the more verbose **function** command, these proof obligations become visible and can be analyzed or solved manually. The expansion from **fun** to **function** is as follows:

```
fun f :: τ                          ≡   function (sequential) f :: τ
where                                   where
  equations                               equations
  ⋮                                       ⋮
                                        by pat_completeness auto
                                        termination by lexicographic_order
```

Some details have now become explicit:

1. The **sequential** option enables the preprocessing of pattern overlaps which we already saw. Without this option, the equations must already be disjoint and complete. The automatic completion only works with constructor patterns.

2. A function definition produces a proof obligation which expresses completeness and compatibility of patterns (we talk about this later). The combination of the methods `pat_completeness` and `auto` is used to solve this proof obligation.

3. A termination proof follows the definition, started by the **termination** command. This will be explained in §4.

Whenever a **fun** command fails, it is usually a good idea to expand the syntax to the more verbose **function** form, to see what is actually going on.

# 4 Termination

The method `lexicographic_order` is the default method for termination proofs. It can prove termination of a certain class of functions by searching for a suitable lexicographic combination of size measures. Of course, not all functions have such a simple termination argument. For them, we can specify the termination relation manually.

## 4.1 The `relation` method

Consider the following function, which sums up natural numbers up to `N`, using a counter `i`:

```
function sum :: "nat ⇒ nat ⇒ nat"
where
  "sum i N = (if i > N then 0 else i + sum (Suc i) N)"
by pat_completeness auto
```

The `lexicographic_order` method fails on this example, because none of the arguments decreases in the recursive call, with respect to the standard size ordering. To prove termination manually, we must provide a custom wellfounded relation.

The termination argument for `sum` is based on the fact that the *difference* between `i` and `N` gets smaller in every step, and that the recursion stops when `i` is greater than `N`. Phrased differently, the expression `N + 1 - i` always decreases.

We can use this expression as a measure function suitable to prove termination.

```
termination sum
apply (relation "measure (λ(i,N). N + 1 - i)")
```

The **termination** command sets up the termination goal for the specified function `sum`. If the function name is omitted, it implicitly refers to the last function definition.

The `relation` method takes a relation of type `('a × 'a) set`, where `'a` is the argument type of the function. If the function has multiple curried arguments, then these are packed together into a tuple, as it happened in the above example.

The predefined function `"measure :: ('a ⇒ nat) ⇒ ('a × 'a) set"` constructs a wellfounded relation from a mapping into the natural numbers (a *measure function*).

After the invocation of `relation`, we must prove that (a) the relation we supplied is wellfounded, and (b) that the arguments of recursive calls indeed decrease with respect to the relation:

```
1. wf (measure (λ(i, N). N + 1 - i))
2. ⋀i N. ¬ N < i ⟹ ((Suc i, N), i, N) ∈ measure (λ(i, N). N + 1 - i)
```

These goals are all solved by `auto`:

```
apply auto
done
```

Let us complicate the function a little, by adding some more recursive calls:

```
function foo :: "nat ⇒ nat ⇒ nat"
where
  "foo i N = (if i > N
              then (if N = 0 then 0 else foo 0 (N - 1))
              else i + foo (Suc i) N)"
by pat_completeness auto
```

When `i` has reached `N`, it starts at zero again and `N` is decremented. This corresponds to a nested loop where one index counts up and the other down. Termination can be proved using a lexicographic combination of two measures, namely the value of `N` and the above difference. The `measures` combinator generalizes `measure` by taking a list of measure functions.

```
termination
by (relation "measures [λ(i, N). N, λ(i,N). N + 1 - i]") auto
```

## 4.2 How `lexicographic_order` works

To see how the automatic termination proofs work, let's look at an example where it fails[2]:

```
fun fails :: "nat ⇒ nat list ⇒ nat"
where
  "fails a [] = a"
| "fails a (x#xs) = fails (x + a) (x#xs)"
```

Isabelle responds with the following error:

```
*** Unfinished subgoals:
*** (a, 1, <):
***  1. ⋀x. x = 0
*** (a, 1, <=):
***  1. False
*** (a, 2, <):
***  1. False
*** Calls:
*** a) (a, x # xs) -->> (x + a, x # xs)
```

[2]For a detailed discussion of the termination prover, see [1]

```
*** Measures:
*** 1) λx. size (fst x)
*** 2) λx. size (snd x)
*** Result matrix:
***     1  2
*** a:  ?  <=
*** Could not find lexicographic termination order.
*** At command "fun".
```

The key to this error message is the matrix at the bottom. The rows of that matrix correspond to the different recursive calls (In our case, there is just one). The columns are the function's arguments (expressed through different measure functions, which map the argument tuple to a natural number).

The contents of the matrix summarize what is known about argument descents: The second argument has a weak descent (`<=`) at the recursive call, and for the first argument nothing could be proved, which is expressed by `?`. In general, there are the values `<`, `<=` and `?`.

For the failed proof attempts, the unfinished subgoals are also printed. Looking at these will often point to a missing lemma.

### 4.3 The `size_change` method

Some termination goals that are beyond the powers of `lexicographic_order` can be solved automatically by the more powerful `size_change` method, which uses a variant of the size-change principle, together with some other techniques. While the details are discussed elsewhere[3], here are a few typical situations where `lexicographic_order` has difficulties and `size_change` may be worth a try:

- Arguments are permuted in a recursive call.
- Several mutually recursive functions with multiple arguments.
- Unusual control flow (e.g., when some recursive calls cannot occur in sequence).

Loading the theory `Multiset` makes the `size_change` method a bit stronger: it can then use multiset orders internally.

## 5 Mutual Recursion

If two or more functions call one another mutually, they have to be defined in one step. Here are `even` and `odd`:

```
function even :: "nat ⇒ bool"
    and odd  :: "nat ⇒ bool"
where
  "even 0 = True"
| "odd 0 = False"
| "even (Suc n) = odd n"
| "odd (Suc n) = even n"
by pat_completeness auto
```

To eliminate the mutual dependencies, Isabelle internally creates a single function operating on the sum type `nat + nat`. Then, `even` and `odd` are defined as projections. Consequently, termination has to be proved simultaneously for both functions, by specifying a measure on the sum type:

```
termination
by (relation "measure (λx. case x of Inl n ⇒ n | Inr n ⇒ n)") auto
```

We could also have used `lexicographic_order`, which supports mutual recursive termination proofs to a certain extent.

## 5.1 Induction for mutual recursion

When functions are mutually recursive, proving properties about them generally requires simultaneous induction. The induction rule `even_odd.induct` generated from the above definition reflects this.

Let us prove something about `even` and `odd`:

```
lemma even_odd_mod2:
  "even n = (n mod 2 = 0)"
  "odd n = (n mod 2 = 1)"
```

We apply simultaneous induction, specifying the induction variable for both goals, separated by **and**:

```
apply (induct n and n rule: even_odd.induct)
```

We get four subgoals, which correspond to the clauses in the definition of `even` and `odd`:

```
1. even 0 = (0 mod 2 = 0)
2. odd 0 = (0 mod 2 = 1)
3. ⋀n. odd n = (n mod 2 = 1) ⟹ even (Suc n) = (Suc n mod 2 = 0)
4. ⋀n. even n = (n mod 2 = 0) ⟹ odd (Suc n) = (Suc n mod 2 = 1)
```

Simplification solves the first two goals, leaving us with two statements about the `mod` operation to prove:

```
apply simp_all
```

```
1. ⋀n. odd n = (n mod 2 = Suc 0) ⟹ (n mod 2 = Suc 0) = (Suc n mod 2 =
0)
2. ⋀n. even n = (n mod 2 = 0) ⟹ (n mod 2 = 0) = (Suc n mod 2 = Suc 0)
```

These can be handled by Isabelle's arithmetic decision procedures.

```
apply arith
apply arith
done
```

In proofs like this, the simultaneous induction is really essential: Even if we are just interested in one of the results, the other one is necessary to strengthen the induction hypothesis. If we leave out the statement about `odd` and just write `True` instead, the same proof fails:

```
lemma failed_attempt:
  "even n = (n mod 2 = 0)"
```

```
  "True"
apply (induct n rule: even_odd.induct)
```

Now the third subgoal is a dead end, since we have no useful induction hypothesis available:

```
1. even 0 = (0 mod 2 = 0)
2. True
3. ⋀n. True ⟹ even (Suc n) = (Suc n mod 2 = 0)
4. ⋀n. even n = (n mod 2 = 0) ⟹ True
```

**oops**

# 6 General pattern matching

## 6.1 Avoiding automatic pattern splitting

Up to now, we used pattern matching only on datatypes, and the patterns were always disjoint and complete, and if they weren't, they were made disjoint automatically like in the definition of `sep` in §2.1.

This automatic splitting can significantly increase the number of equations involved, and this is not always desirable. The following example shows the problem:

Suppose we are modeling incomplete knowledge about the world by a three-valued datatype, which has values `T`, `F` and `X` for true, false and uncertain propositions, respectively.

```
datatype P3 = T | F | X
```

Then the conjunction of such values can be defined as follows:

```
fun And :: "P3 ⇒ P3 ⇒ P3"
where
  "And T p = p"
| "And p T = p"
| "And p F = F"
| "And F p = F"
| "And X X = X"
```

This definition is useful, because the equations can directly be used as simplification rules. But the patterns overlap: For example, the expression `And T T` is matched by both the first and the second equation. By default, Isabelle makes the patterns disjoint by splitting them up, producing instances:

```
thm And.simps
```

```
And T ?p = ?p
And F T = F
And X T = X
And F F = F
And X F = F
And F X = F
And X X = X
```

There are several problems with this:

1. If the datatype has many constructors, there can be an explosion of equations. For `And`, we get seven instead of five equations, which can be tolerated, but this is just a small example.

2. Since splitting makes the equations "less general", they do not always match in rewriting. While the term `And x F` can be simplified to `F` with the original equations, a (manual) case split on `x` is now necessary.

3. The splitting also concerns the induction rule `And.induct`. Instead of five premises it now has seven, which means that our induction proofs will have more cases.

4. In general, it increases clarity if we get the same definition back which we put in.

If we do not want the automatic splitting, we can switch it off by leaving out the **sequential** option. However, we will have to prove that our pattern matching is consistent[3]:

```
function And2 :: "P3 ⇒ P3 ⇒ P3"
where
  "And2 T p = p"
| "And2 p T = p"
| "And2 p F = F"
| "And2 F p = F"
| "And2 X X = X"
```

Now let's look at the proof obligations generated by a function definition. In this case, they are:

```
1. ⋀P x. ⟦⋀p. x = (T, p) ⟹ P; ⋀p. x = (p, T) ⟹ P; ⋀p. x = (p, F)
⟹ P;
            ⋀p. x = (F, p) ⟹ P; x = (X, X) ⟹ P⟧
           ⟹ P
 2. ⋀p pa. (T, p) = (T, pa) ⟹ p = pa
 3. ⋀p pa. (T, p) = (pa, T) ⟹ p = pa
 4. ⋀p pa. (T, p) = (pa, F) ⟹ p = F
 5. ⋀p pa. (T, p) = (F, pa) ⟹ p = F
 6. ⋀p. (T, p) = (X, X) ⟹ p = X
 7. ⋀p pa. (p, T) = (pa, T) ⟹ p = pa
 8. ⋀p pa. (p, T) = (pa, F) ⟹ p = F
 9. ⋀p pa. (p, T) = (F, pa) ⟹ p = F
10. ⋀p. (p, T) = (X, X) ⟹ p = X
                  ⋮
```

The first subgoal expresses the completeness of the patterns. It has the form of an elimination rule and states that every `x` of the function's input type must match at least one of the patterns[4]. If the patterns just involve datatypes, we can solve it with the `pat_completeness` method:

[3]This prevents us from defining something like `f x = True` and `f x = False` simultaneously.

[4]Completeness could be equivalently stated as a disjunction of existential statements: `(∃p. x = (T, p)) ∨ (∃p. x = (p, T)) ∨ (∃p. x = (p, F)) ∨ (∃p. x = (F, p)) ∨ x = (X, X)`, and you can use the method `atomize_elim` to get that form instead.

**apply** `pat_completeness`

The remaining subgoals express *pattern compatibility*. We do allow that an input value matches multiple patterns, but in this case, the result (i.e. the right hand sides of the equations) must also be equal. For each pair of two patterns, there is one such subgoal. Usually this needs injectivity of the constructors, which is used automatically by `auto`.

**by** `auto`

## 6.2 Non-constructor patterns

Most of Isabelle's basic types take the form of inductive datatypes, and usually pattern matching works on the constructors of such types. However, this need not be always the case, and the **function** command handles other kind of patterns, too.

One well-known instance of non-constructor patterns are so-called $n + k$-*patterns*, which are a little controversial in the functional programming world. Here is the initial fibonacci example with $n + k$-patterns:

```
function fib2 :: "nat ⇒ nat"
where
  "fib2 0 = 1"
| "fib2 1 = 1"
| "fib2 (n + 2) = fib2 n + fib2 (Suc n)"
```

This kind of matching is again justified by the proof of pattern completeness and compatibility. The proof obligation for pattern completeness states that every natural number is either 0, 1 or `n + 2`:

```
1. ⋀P x. ⟦x = 0 ⟹ P; x = 1 ⟹ P; ⋀n. x = n + 2 ⟹ P⟧ ⟹ P
```

This is an arithmetic triviality, but unfortunately the `arith` method cannot handle this specific form of an elimination rule. However, we can use the method `atomize_elim` to do an ad-hoc conversion to a disjunction of existentials, which can then be solved by the arithmetic decision procedure. Pattern compatibility and termination are automatic as usual.

```
apply atomize_elim
apply arith
apply auto
done
termination by lexicographic_order
```

We can stretch the notion of pattern matching even more. The following function is not a sensible functional program, but a perfectly valid mathematical definition:

```
function ev :: "nat ⇒ bool"
where
  "ev (2 * n) = True"
| "ev (2 * n + 1) = False"
apply atomize_elim
by arith+
termination by (relation "{}") simp
```

This general notion of pattern matching gives you a certain freedom in writing down specifications. However, as always, such freedom should be used with care:

If we leave the area of constructor patterns, we have effectively departed from the world of functional programming. This means that it is no longer possible to use the code generator, and expect it to generate ML code for our definitions. Also, such a specification might not work very well together with simplification. Your mileage may vary.

## 6.3 Conditional equations

The function package also supports conditional equations, which are similar to guards in a language like Haskell. Here is Euclid's algorithm written with conditional patterns[5]:

```
function gcd :: "nat ⇒ nat ⇒ nat"
where
  "gcd x 0 = x"
| "gcd 0 y = y"
| "x < y ⟹ gcd (Suc x) (Suc y) = gcd (Suc x) (y - x)"
| "¬ x < y ⟹ gcd (Suc x) (Suc y) = gcd (x - y) (Suc y)"
by (atomize_elim, auto, arith)
termination by lexicographic_order
```

By now, you can probably guess what the proof obligations for the pattern completeness and compatibility look like.

Again, functions with conditional patterns are not supported by the code generator.

## 6.4 Pattern matching on strings

As strings (as lists of characters) are normal datatypes, pattern matching on them is possible, but somewhat problematic. Consider the following definition:

```
fun check :: "string ⇒ bool"
where
  "check (''good'') = True"
| "check s = False"
```

An invocation of the above **fun** command does not terminate. What is the problem? Strings are lists of characters, and characters are a datatype with a lot of constructors. Splitting the catch-all pattern thus leads to an explosion of cases, which cannot be handled by Isabelle.

There are two things we can do here. Either we write an explicit `if` on the right hand side, or we can use conditional patterns:

```
function check :: "string ⇒ bool"
where
  "check (''good'') = True"
| "s ≠ ''good'' ⟹ check s = False"
by auto
```

[5] Note that the patterns are also overlapping in the base case

# 7 Partiality

In HOL, all functions are total. A function `f` applied to `x` always has the value `f x`, and there is no notion of undefinedness. This is why we have to do termination proofs when defining functions: The proof justifies that the function can be defined by wellfounded recursion.

However, the **function** package does support partiality to a certain extent. Let's look at the following function which looks for a zero of a given function f.

```
function findzero :: "(nat ⇒ nat) ⇒ nat ⇒ nat"
where
  "findzero f n = (if f n = 0 then n else findzero f (Suc n))"
by pat_completeness auto
```

Clearly, any attempt of a termination proof must fail. And without that, we do not get the usual rules `findzero.simps` and `findzero.induct`. So what was the definition good for at all?

## 7.1 Domain predicates

The trick is that Isabelle has not only defined the function `findzero`, but also a predicate `findzero_dom` that characterizes the values where the function terminates: the *domain* of the function. If we treat a partial function just as a total function with an additional domain predicate, we can derive simplification and induction rules as we do for total functions. They are guarded by domain conditions and are called `psimps` and `pinduct`:

```
findzero_dom (?f, ?n) ⟹
findzero ?f ?n = (if ?f ?n = 0 then ?n else findzero ?f
(Suc ?n))                                                    (findzero.psimps)
```

```
⟦findzero_dom (?a0.0, ?a1.0);
 ⋀f n. ⟦findzero_dom (f, n); f n ≠ 0 ⟹ ?P f (Suc n)⟧
⟹ ?P f n⟧                                                    (findzero.pinduct)
⟹ ?P ?a0.0 ?a1.0
```

Remember that all we are doing here is use some tricks to make a total function appear as if it was partial. We can still write the term `findzero (λx. 1) 0` and like any other term of type `nat` it is equal to some natural number, although we might not be able to find out which one. The function is *underdefined*.

But it is defined enough to prove something interesting about it. We can prove that if `findzero f n` terminates, it indeed returns a zero of `f`:

```
lemma findzero_zero: "findzero_dom (f, n) ⟹ f (findzero f n) = 0"
```

We apply induction as usual, but using the partial induction rule:

```
apply (induct f n rule: findzero.pinduct)
```

This gives the following subgoals:

```
1. ⋀f n. ⟦findzero_dom (f, n); f n ≠ 0 ⟹ f (findzero f (Suc n)) = 0⟧
         ⟹ f (findzero f n) = 0
```

The hypothesis in our lemma was used to satisfy the first premise in the induction rule. However, we also get `findzero_dom (f, n)` as a local assumption in

```
lemma ⟦findzero-dom (f, n); x ∈ {n ..< findzero f n}⟧ ⟹ f x ≠ 0
proof (induct rule: findzero.pinduct)
  fix f n assume dom: findzero-dom (f, n)
             and IH: ⟦f n ≠ 0; x ∈ {Suc n ..< findzero f (Suc n)}⟧
⟹ f x ≠ 0
             and x-range: x ∈ {n ..< findzero f n}
  have f n ≠ 0
  proof
    assume f n = 0
    with dom have findzero f n = n by (simp add: findzero.psimps)
    with x-range show False by auto
  qed

  from x-range have x = n ∨ x ∈ {Suc n ..< findzero f n} by auto
  thus f x ≠ 0
  proof
    assume x = n
    with ⟨f n ≠ 0⟩ show ?thesis by simp
  next
    assume x ∈ {Suc n ..< findzero f n}
    with dom and ⟨f n ≠ 0⟩ have x ∈ {Suc n ..< findzero f (Suc
n)} by (simp add: findzero.psimps)
    with IH and ⟨f n ≠ 0⟩
    show ?thesis by simp
  qed
qed
```

Figure 1: A proof about a partial function

the induction step. This allows unfolding `findzero f n` using the `psimps` rule, and the rest is trivial.

```
apply (simp add: findzero.psimps)
done
```

Proofs about partial functions are often not harder than for total functions. Fig. 1 shows a slightly more complicated proof written in Isar. It is verbose enough to show how partiality comes into play: From the partial induction, we get an additional domain condition hypothesis. Observe how this condition is applied when calls to `findzero` are unfolded.

## 7.2 Partial termination proofs

Now that we have proved some interesting properties about our function, we should turn to the domain predicate and see if it is actually true for some values. Otherwise we would have just proved lemmas with `False` as a premise.

Essentially, we need some introduction rules for `findzero_dom`. The function package can prove such domain introduction rules automatically. But since they are not used very often (they are almost never needed if the function is total), this functionality is disabled by default for efficiency reasons. So we have to go back and ask for them explicitly by passing the `(domintros)` option to the function package:

```
lemma findzero-termination:
  assumes x ≥ n and f x = 0
  shows findzero-dom (f, n)
proof −
  have base: findzero-dom (f, x)
    by (rule findzero.domintros) (simp add:⟨f x = 0⟩)

  have step: ⋀i. findzero-dom (f, Suc i)
    ⟹ findzero-dom (f, i)
    by (rule findzero.domintros) simp

  from ⟨x ≥ n⟩ show ?thesis
  proof (induct rule:inc-induct)
    show findzero-dom (f, x) by (rule base)
  next
    fix i assume findzero-dom (f, Suc i)
    thus findzero-dom (f, i) by (rule step)
  qed
qed
```

Figure 2: Termination proof for `findzero`

```
function (domintros) findzero :: "(nat ⇒ nat) ⇒ nat ⇒ nat"
where
  ...
```

Now the package has proved an introduction rule for `findzero_dom`:

```
thm findzero.domintros
```

```
(0 < ?f ?n ⟹ findzero_dom (?f, Suc ?n)) ⟹ findzero_dom (?f, ?n)
```

Domain introduction rules allow to show that a given value lies in the domain of a function, if the arguments of all recursive calls are in the domain as well. They allow to do a "single step" in a termination proof. Usually, you want to combine them with a suitable induction principle.

Since our function increases its argument at recursive calls, we need an induction principle which works "backwards". We will use `inc_induct`, which allows to do induction from a fixed number "downwards":

```
⟦?i ≤ ?j; ?P ?j; ⋀i. ⟦i < ?j; ?P (Suc i)⟧ ⟹ ?P i⟧ ⟹ ?P ?i
```
(`inc_induct`)

Figure 2 gives a detailed Isar proof of the fact that `findzero` terminates if there is a zero which is greater or equal to `n`. First we derive two useful rules which will solve the base case and the step case of the induction. The induction is then straightforward, except for the unusual induction principle.

Again, the proof given in Fig. 2 has a lot of detail in order to explain the principles. Using more automation, we can also have a short proof:

```
lemma findzero_termination_short:
  assumes zero: "x >= n"
```

```
  assumes [simp]: "f x = 0"
  shows "findzero_dom (f, n)"
using zero
by (induct rule:inc_induct) (auto intro: findzero.domintros)
```

It is simple to combine the partial correctness result with the termination lemma:

```
lemma findzero_total_correctness:
  "f x = 0 ⟹ f (findzero f 0) = 0"
by (blast intro: findzero_zero findzero_termination)
```

## 7.3 Definition of the domain predicate

Sometimes it is useful to know what the definition of the domain predicate looks like. Actually, `findzero_dom` is just an abbreviation:

```
findzero_dom ≡ accp findzero_rel
```

The domain predicate is the *accessible part* of a relation `findzero_rel`, which was also created internally by the function package. `findzero_rel` is just a normal inductive predicate, so we can inspect its definition by looking at the introduction rules `findzero_rel.intros`. In our case there is just a single rule:

```
?f ?n ≠ 0 ⟹ findzero_rel (?f, Suc ?n) (?f, ?n)
```

The predicate `findzero_rel` describes the *recursion relation* of the function definition. The recursion relation is a binary relation on the arguments of the function that relates each argument to its recursive calls. In general, there is one introduction rule for each recursive call.

The predicate `findzero_dom` is the accessible part of that relation. An argument belongs to the accessible part, if it can be reached in a finite number of steps (cf. its definition in `Wellfounded.thy`).

Since the domain predicate is just an abbreviation, you can use lemmas for `accp` and `findzero_rel` directly. Some lemmas which are occasionally useful are `accpI`, `accp_downward`, and of course the introduction and elimination rules for the recursion relation `findzero.intros` and `findzero.cases`.

## 7.4 A Useful Special Case: Tail recursion

The domain predicate is our trick that allows us to model partiality in a world of total functions. The downside of this is that we have to carry it around all the time. The termination proof above allowed us to replace the abstract `findzero_dom (f, n)` by the more concrete `n ≤ x ∧ f x = 0`, but the condition is still there and can only be discharged for special cases. In particular, the domain predicate guards the unfolding of our function, since it is there as a condition in the `psimp` rules.

Now there is an important special case: We can actually get rid of the condition in the simplification rules, *if the function is tail-recursive*. The reason is that for all tail-recursive equations there is a total function satisfying them, even if they are non-terminating.

The function package internally does the right construction and can derive the unconditional simp rules, if we ask it to do so. Luckily, our `findzero` function

is tail-recursive, so we can just go back and add another option to the **function** command:

```
function (domintros, tailrec) findzero :: "(nat ⇒ nat) ⇒ nat ⇒ nat"
where
  ...
```

Now, we actually get unconditional simplification rules, even though the function is partial:

```
thm findzero.simps
```

```
findzero ?f ?n = (if ?f ?n = 0 then ?n else findzero ?f (Suc ?n))
```

Of course these would make the simplifier loop, so we better remove them from the simpset:

```
declare findzero.simps[simp del]
```

Getting rid of the domain conditions in the simplification rules is not only useful because it simplifies proofs. It is also required in order to use Isabelle's code generator to generate ML code from a function definition. Since the code generator only works with equations, it cannot be used with `psimp` rules. Thus, in order to generate code for partial functions, they must be defined as a tail recursion. Luckily, many functions have a relatively natural tail recursive definition.

# 8 Nested recursion

Recursive calls which are nested in one another frequently cause complications, since their termination proof can depend on a partial correctness property of the function itself.

As a small example, we define the "nested zero" function:

```
function nz :: "nat ⇒ nat"
where
  "nz 0 = 0"
| "nz (Suc n) = nz (nz n)"
by pat_completeness auto
```

If we attempt to prove termination using the identity measure on naturals, this fails:

```
termination
  apply (relation "measure (λn. n)")
  apply auto
```

We get stuck with the subgoal

```
1. ⋀n. nz_dom n ⟹ nz n < Suc n
```

Of course this statement is true, since we know that `nz` is the zero function. And in fact we have no problem proving this property by induction.

```
lemma nz_is_zero: "nz_dom n ⟹ nz n = 0"
```

**function** $f91 :: nat \Rightarrow nat$
**where**
$f91\ n = (if\ 100 < n\ then\ n - 10\ else\ f91\ (f91\ (n + 11)))$
**by** *pat-completeness auto*

**lemma** $f91$-*estimate*:
**assumes** *trm*: $f91$-*dom* $n$
**shows** $n < f91\ n + 11$
**using** *trm* **by** *induct* (*auto simp*: $f91$.*psimps*)

**termination**
**proof**
**let** *?R* = *measure* $(\lambda x.\ 101 - x)$
**show** *wf ?R* **..**

**fix** $n :: nat$ **assume** $\neg\ 100 < n$ — Assumptions for both calls

**thus** $(n + 11,\ n) \in$ *?R* **by** *simp* — Inner call

**assume** *inner-trm*: $f91$-*dom* $(n + 11)$ — Outer call
**with** $f91$-*estimate* **have** $n + 11 < f91\ (n + 11) + 11$ **.**
**with** $\langle \neg\ 100 < n \rangle$ **show** $(f91\ (n + 11),\ n) \in$ *?R* **by** *simp*
**qed**

Figure 3: McCarthy's 91-function

```
  by (induct rule:nz.pinduct) (auto simp: nz.psimps)
```

We formulate this as a partial correctness lemma with the condition `nz_dom n`. This allows us to prove it with the `pinduct` rule before we have proved termination. With this lemma, the termination proof works as expected:

```
termination
  by (relation "measure (λn. n)") (auto simp: nz_is_zero)
```

As a general strategy, one should prove the statements needed for termination as a partial property first. Then they can be used to do the termination proof. This also works for less trivial examples. Figure 3 defines the 91-function, a well-known challenge problem due to John McCarthy, and proves its termination.

# 9 Higher-Order Recursion

Higher-order recursion occurs when recursive calls are passed as arguments to higher-order combinators such as `map`, `filter` etc. As an example, imagine a datatype of n-ary trees:

```
datatype 'a tree =
  Leaf 'a
| Branch "'a tree list"
```

We can define a function which swaps the left and right subtrees recursively, using the list functions `rev` and `map`:

```
fun mirror :: "'a tree ⇒ 'a tree"
where
  "mirror (Leaf n) = Leaf n"
| "mirror (Branch l) = Branch (rev (map mirror l))"
```

Although the definition is accepted without problems, let us look at the termination proof:

**termination proof**

As usual, we have to give a wellfounded relation, such that the arguments of the recursive calls get smaller. But what exactly are the arguments of the recursive calls when mirror is given as an argument to `map`? Isabelle gives us the subgoals

```
1. wf ?R
2. ⋀l x. x ∈ set l ⟹ (x, Branch l) ∈ ?R
```

So the system seems to know that `map` only applies the recursive call `mirror` to elements of `l`, which is essential for the termination proof.

This knowledge about `map` is encoded in so-called congruence rules, which are special theorems known to the **function** command. The rule for `map` is

```
⟦?xs = ?ys; ⋀x. x ∈ set ?ys ⟹ ?f x = ?g x⟧ ⟹ map ?f ?xs = map ?g ?ys
```

You can read this in the following way: Two applications of `map` are equal, if the list arguments are equal and the functions coincide on the elements of the list. This means that for the value `map f l` we only have to know how `f` behaves on the elements of `l`.

Usually, one such congruence rule is needed for each higher-order construct that is used when defining new functions. In fact, even basic functions like `If` and `Let` are handled by this mechanism. The congruence rule for `If` states that the `then` branch is only relevant if the condition is true, and the `else` branch only if it is false:

```
⟦?b = ?c; ?c ⟹ ?x = ?u; ¬ ?c ⟹ ?y = ?v⟧
⟹ (if ?b then ?x else ?y) = (if ?c then ?u else ?v)
```

Congruence rules can be added to the function package by giving them the `fundef_cong` attribute.

The constructs that are predefined in Isabelle, usually come with the respective congruence rules. But if you define your own higher-order functions, you may have to state and prove the required congruence rules yourself, if you want to use your functions in recursive definitions.

## 9.1 Congruence Rules and Evaluation Order

Higher order logic differs from functional programming languages in that it has no built-in notion of evaluation order. A program is just a set of equations, and it is not specified how they must be evaluated.

However for the purpose of function definition, we must talk about evaluation order implicitly, when we reason about termination. Congruence rules express that a certain evaluation order is consistent with the logical definition.

Consider the following function.

```
function f :: "nat ⇒ bool"
where
  "f n = (n = 0 ∨ f (n - 1))"
```

For this definition, the termination proof fails. The default configuration specifies no congruence rule for disjunction. We have to add a congruence rule that specifies left-to-right evaluation order:

```
⟦?P = ?P'; ¬ ?P' ⟹ ?Q = ?Q'⟧ ⟹ (?P ∨ ?Q) = (?P' ∨ ?Q')    (disj_cong)
```

Now the definition works without problems. Note how the termination proof depends on the extra condition that we get from the congruence rule.

However, as evaluation is not a hard-wired concept, we could just turn everything around by declaring a different congruence rule. Then we can make the reverse definition:

```
lemma disj_cong2[fundef_cong]:
  "(¬ Q' ⟹ P = P') ⟹ (Q = Q') ⟹ (P ∨ Q) = (P' ∨ Q')"
  by blast

fun f' :: "nat ⇒ bool"
where
  "f' n = (f' (n - 1) ∨ n = 0)"
```

These examples show that, in general, there is no "best" set of congruence rules.

However, such tweaking should rarely be necessary in practice, as most of the time, the default set of congruence rules works well.

# References


[1] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, 2007.

[2] Alexander Krauss. Partial recursive functions in Higher-Order Logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning: IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer-Verlag, 2006.

[3] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2009.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[5] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.