Isabelle: Recursión e inducción

Francisco J. Martín Mateos

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Principio de definición

- La lógica se extiende dinámicamente, añadiendo nuevas definiciones de funciones usando la directiva `fun` entre otras posibilidades.
  - Cada definición añade nuevos axiomas a la lógica
  - La consistencia de la lógica se debe mantener
- Sólo se admiten funciones bajo ciertas restricciones
  - Sólo se puede hacer referencia a funciones previamente definidas
  - Las únicas variables de la definición han de ser las que aparecen como sus argumentos o las introducidas mediante entornos locales
  - Sólo se admiten funciones que terminan para cualquier dato de entrada
- Esto asegura extensiones consistentes y conservativas de la lógica

Principio de definición

- Dada una historia $\mathcal{H}$ (secuencia de definiciones previas), una definición por casos con la directiva `fun` es admisible respecto de $\mathcal{H}$ si
  - se introduce un símbolo de función `f` nuevo (no aparece en $\mathcal{H}$) de aridad **n**,
  - los casos de la definición de `f` son una partición completa del dominio declarado de la función `f`,
  - el cuerpo de cada caso de la definición es un término en el lenguaje de $\mathcal{H}$ ampliado con el símbolo `f` de aridad **n**, cuyas variables libres están entre las usadas para especificar el caso,
  - y se pueden demostrar las *conjeturas de terminación* de `f`
- Si la definición es aceptada, para cada caso de la definición de `f` se introduce un axioma que expresa la regla de sustitución asociada.

Principio de definición

- Definición de la función fibonacci

```
fun fib0 :: "nat => nat" where
  "fib0 0 = 1"
| "fib0 (suc 0) = 1"
| "fib0 (Suc (Suc n)) = fib0 n + fib0 (Suc n)"
```

- Definición con un entorno local

```
fun fib1 :: "nat => nat" where
  "fib1 0 = 1"
| "fib1 (Suc 0) = 1"
| "fib1 (Suc (Suc n)) = (let m = n+1 in fib1 n + fib1 m)"
```

Principio de definición

- No se permite el uso de variables globales

```
fun fib2 :: "nat => nat" where
  "fib2 0 = y"
| "fib2 (suc 0) = y"
| "fib2 (Suc (Suc n)) = fib2 n + fib2 (Suc n)"
```

- Esta definición añadiría el siguiente axioma a la lógica:

$$\texttt{fib2 0} = \texttt{y}$$

donde `y` es una variable global, cuyo valor puede cambiar después de la definición de `fib2`. Este cambio de valor puede hacer que el axioma anterior produzca una inconsistencia en el sistema

Principio de definición

- No se permiten las recursiones infinitas

```
fun infinita :: "nat => nat" where
  "infinita n = 1 + (infinita n)"
```

- Esta definición añadiría el siguiente axioma a la lógica:

$$\texttt{infinita n} = \texttt{1 + (infinita n)}$$

- El uso de esta igualdad como regla de simplificación introduciría el sistema en un bucle infinito

Principio de definición

- No se permiten definiciones por recursión mútua

```
fun par :: "nat => bool" where
  "par 0 = True"
| "par (Suc n) = impar n"

fun impar :: "nat => bool" where
  "impar 0 = False"
| "impar (Suc n) = par n"
```

- La primera definición incluiría un axioma que hace referencia a un símbolo de función desconocido !!

Principio de definición

- Pero sí la definición de dos funciones distintas en la misma directiva `fun`

```
fun par :: "nat => bool"
and impar :: "nat => bool" where
  "par 0 = True"
| "par (Suc n) = impar n"
| "impar 0 = False"
| "impar (Suc n) = par n"
```

Principio de definición

- El propio sistema fuerza a que las funciones tomen un valor para cualquier dato de entrada (funciones totales)
  - Comprueba que los casos de la definición forman una partición completa y consistente del dominio de la función definida
  - Las funciones predefinidas en el sistema son totales (salvo contadas excepciones que generan errores controlados)
  - Sólo se permite la versión completa del condicional `if`

Principio de definición

- El sistema es capaz de detectar cuando faltan casos en una definición

```
fun fib3 :: "nat => nat" where
  "fib3 0 = 1"
| "fib3 (Suc 0) = 1"
```

- Aunque no se produce un error, el sistema identifica la situación en la que la definición no se puede usar.

Principio de definición

- Podemos ver detalles sobre las conjeturas asociadas a los casos de la definición con la directiva ampliada **`function`**

```
function fib4 :: "nat => nat" where
  "fib4 0 = 1"
| "fib4 (Suc 0) = 1"
| "fib4 (Suc (Suc n)) = fib4 n + fib4 (Suc n)"
  by pat_completeness auto
```

- Al usar esta directiva hay que indicar las estrategias que se deben usar para demostrar las propiedades de completitud
  - **`pat_completeness`** es una estrategia que demuestra de forma automática la completitud de los casos de la definición

Conjeturas de terminación

- Dada una historia $\mathcal{H}$ (secuencia de definiciones previas), las conjeturas de terminación de una definición aseguran que existe una *relación* R y una *medida* en el lenguaje de $\mathcal{H}$ verificando
  - La *medida* justifica que la relación R es bien fundamentada
    - No existen cadenas descendentes infinitas
  - En cada llamada recursiva, si se cumplen los términos que influyen en dicha llamada entonces los argumentos de la llamada recursiva están relacionados con respecto a R con los argumentos del caso correspondiente, es decir, son menores con respecto a R

Admisión de funciones recursivas

- La función factorial

```
fun factorial :: "nat => nat" where
  "factorial 0 = 1"
| "factorial (Suc n) = n * factorial n"
```

  - La relación podría ser la descrita por el conjunto $\mathbf{R} = \{(\mathbf{n}, \mathbf{Suc\ n}), \mathbf{n} \in \mathbb{N}\}$
    - La medida que justifica la buena fundamentación es la función identidad
  - Hay una única llamada recursiva: `factorial n`
    - La conjetura de terminación es $(\mathbf{n}, \mathbf{Suc\ n}) \in \mathbf{R}$

Admisión de funciones recursivas

- En una recursión infinita no es posible encontrar una relación bien fundamentada con respecto a la cual los argumentos de la llamada recursiva sean menores que los argumentos del original

```
fun infinita :: "nat => nat" where
  "infinita n = 1 + (infinita n)"
```

  - En este caso el argumento de la llamada recursiva es el mismo que el original, por tanto es imposible que sea menor con respecto a una relación buen fundamentada

Admisión de funciones recursivas

- Concatenación de listas

```
fun concatena :: "'a list => 'a list => 'a list" where
  "concatena [] ys = ys"
| "concatena (x#xs) ys = x#(concatena xs ys)"
```

  - La relación podría ser la descrita por el conjunto
    $\mathbf{R = \{((x1s, y1s), (x2s, y2s)), length\ x1s < length\ x2s\}}$
    - La medida que justifica la buena fundamentación es la longitud de la primera componente
  - Hay una única llamada recursiva: `concatena xs ys`
    - La conjetura de terminación es $\mathbf{((xs, ys), (x\#xs, ys)) \in R}$

Admisión de funciones recursivas

- Listas ordenadas

```
fun ordenada :: "nat list => bool" where
  "ordenada [] = True"
| "ordenada [x] = True"
| "ordenada (x1#x2#xs) = (x1 <= x2 /\ ordenada (x2#xs))"
```

  - La relación podría ser la descrita por el conjunto $\mathbf{R} = \{(\mathbf{xs}, \mathbf{ys}), \mathbf{length\ xs} < \mathbf{length\ ys}\}$
    - La medida que justifica la buena fundamentación es la función longitud
  - Hay una única llamada recursiva: `ordenada (x2#xs)`
    - La conjetura de terminación es $(\mathbf{x2\#xs}, \mathbf{x1\#x2\#xs}) \in \mathbf{R}$

Admisión de funciones recursivas

- Mezcla ordenada de dos listas

```
fun mezcla :: "nat list => nat list => nat list" where
  "mezcla [] ys = ys"
| "mezcla xs [] = xs"
| "mezcla (x#xs) (y#ys) = (if x < y
                           then x#(mezcla xs (y#ys))
                           else y#(mezcla (x#xs) ys))"
```

  - La relación podría ser la descrita por el conjunto
    $\mathbf{R} = \{((\mathbf{x1s}, \mathbf{y1s}), (\mathbf{x2s}, \mathbf{y2s})),$
    $\mathbf{length\ x1s} + \mathbf{length\ y1s} < \mathbf{length\ x2s} + \mathbf{length\ y2s}\}$
    - La medida que justifica la buena fundamentación es la suma de las longitudes de las dos componentes
  - La primera llamada recursiva es: `mezcla xs (y#ys)`
    - La conjetura de terminación es
      $\mathbf{x} < \mathbf{y} \rightarrow ((\mathbf{xs}, \mathbf{y\#ys}), (\mathbf{x\#xs}, \mathbf{y\#ys})) \in \mathbf{R}$
  - La segunda llamada recursiva es: `mezcla (x#xs) ys`
    - La conjetura de terminación es
      $\neg(\mathbf{x} < \mathbf{y}) \rightarrow ((\mathbf{x\#xs}, \mathbf{ys}), (\mathbf{x\#xs}, \mathbf{y\#ys})) \in \mathbf{R}$

Admisión de funciones recursivas

- Mezcla ordenada de dos listas

```
fun mezcla :: "nat list => nat list => nat list" where
  "mezcla [] ys = ys"
| "mezcla xs [] = xs"
| "mezcla (x#xs) (y#ys) = (if x < y
                           then x#(mezcla xs (y#ys))
                           else y#(mezcla (x#xs) ys))"
```

  - También se podría definir la relación a partir del orden lexicográfico de varias medidas
    $\mathbf{R} = \{((\mathbf{x1s}, \mathbf{y1s}), (\mathbf{x2s}, \mathbf{y2s})),$
    $\mathbf{length\ x1s} < \mathbf{length\ x2s} \vee \mathbf{length\ y1s} < \mathbf{length\ y2s}\}$
    - El orden lexicográfico compara primero con respecto a la primera medida y, en caso de que sean iguales, pasa a comparar con respecto a la siguiente

El principio de inducción estructural

- Una prueba por inducción de $\forall \mathbf{n} \in \mathbb{N}, \phi(\mathbf{n})$, con

$$\phi(\mathbf{n}) \equiv \mathbf{n}^2 = \sum_{\mathbf{k=0}}^{\mathbf{n-1}} (\mathbf{2k+1})$$

- Caso base: $\mathbf{n} \notin \mathbb{N}^+ \rightarrow \phi(\mathbf{n})$
  - La propiedad no tiene validez cuando $\mathbf{n} \notin \mathbb{N}$
  - La propiedad es trivial para $\mathbf{n = 0}$
- Caso de inducción: $(\mathbf{n} \in \mathbb{N}^+) \wedge \phi(\mathbf{n-1}) \rightarrow \phi(\mathbf{n})$
  - La inducción es válida pues la hipótesis de inducción se establece para el valor $\mathbf{n-1}$, menor que $\mathbf{n}$, y el conjunto de los números naturales está bien ordenado con respecto al orden usual

El principio de inducción estructural

- La fórmula $\forall \bar{x}\ \phi(\bar{x})$ se deriva a partir de las siguientes fórmulas
  - *Casos de inducción*: Para cada $1 \leq i \leq k$

$$q_i(\bar{x}) \wedge \sigma_{i,1}(\phi(\bar{x})) \wedge \ldots \wedge \sigma_{i,h_i}(\phi(\bar{x})) \rightarrow \phi(\bar{x})$$

  - *Caso base*:

$$\neg q_1(\bar{x}) \wedge \ldots \wedge \neg q_k(\bar{x}) \rightarrow \phi(\bar{x})$$

donde $q_1(\bar{x}),\ldots,q_k(\bar{x})$ son términos, $\sigma_{i,j}$, $(\forall i, j)$ son sustituciones y existe un término $m(\bar{x})$ y una relación bien fundamentada $<_R$ tal que para cada i, j, se tiene

$$q_i(\bar{x}) \rightarrow \sigma_{i,j}(m(\bar{x})) <_R m(\bar{x})$$

El principio de inducción estructural

- Una prueba por inducción de $\forall \mathbf{n} \in \mathbb{N}, \phi(\mathbf{n})$, con

$$\phi(\mathbf{n}) \equiv \mathbf{n}^2 = \sum_{\mathbf{k}=\mathbf{0}}^{\mathbf{n}-\mathbf{1}} (\mathbf{2k}+\mathbf{1})$$

  - Caso de inducción: $\mathbf{q_1(n)} \equiv (\mathbf{n} \in \mathbb{N}^+)$, $\sigma_{1,1} = \{\mathbf{n}/\mathbf{n}-\mathbf{1}\}$

$$\mathbf{q_1(n)} \wedge \sigma_{1,1}(\phi(\mathbf{n})) \rightarrow \phi(\mathbf{n})$$

  - Caso base:

$$\neg \mathbf{q_1(n)} \rightarrow \phi(\mathbf{n})$$

- El término $\mathbf{m(n)}$ es $\mathbf{n}$ y la relación $<_{\mathbf{R}}$ es el orden habitual entre los números naturales

$$\mathbf{q_1(n)} \rightarrow \sigma_{1,1}(\mathbf{m(n)}) <_{\mathbf{R}} \mathbf{m(n)}$$

El principio de inducción estructural

- Una prueba por inducción de $\forall \mathbf{n} \in \mathbb{N}, \phi(\mathbf{n})$, con

$$\phi(\mathbf{n}) \equiv \mathbf{n}^2 = \sum_{\mathbf{k}=\mathbf{0}}^{\mathbf{n}-\mathbf{1}} (\mathbf{2k}+\mathbf{1})$$

  - Caso de inducción: $\mathbf{q_1(n)} \equiv (\mathbf{n} \in \mathbb{N}^+)$, $\sigma_{1,1} = \{\mathbf{n}/\mathbf{n}-\mathbf{1}\}$

$$\mathbf{n} \in \mathbb{N}^+ \wedge \phi(\mathbf{n}-\mathbf{1}) \rightarrow \phi(\mathbf{n})$$

  - Caso base:

$$\phi(\mathbf{0})$$

- El término $\mathbf{m(n)}$ es $\mathbf{n}$ y la relación $<_{\mathbf{R}}$ es el orden habitual entre los números naturales

$$\mathbf{n} \in \mathbb{N}^+ \rightarrow \mathbf{n}-\mathbf{1} < \mathbf{n}$$

El principio de inducción estructural

- Definición de la función **sumaImpares**

```
fun sumaImpares :: "nat => nat" where
  "sumaImpares 0 = 0"
| "sumaImpares (Suc n) = (2*n+1) + sumaImpares n"
```

- Propiedad a demostrar:

```
lemma sumaImparesCuadrado:
  "sumaImpares n = n^2"
```

- Objetivos generados con **(induct n)**

```
1. sumaImpares 0 = 0^2
2. /\n. sumaImpares n = n^2 ==>
        sumaImpares (Suc n) = (Suc n)^2
```

Inducción asociada a tipos de datos recursivos

- Números naturales

```
datatype nat =
  0
| Suc nat

thm nat.induct
```

- Esquema de inducción asociado

```
?P 0 ==>
(/\nat. ?P nat ==> ?P (Suc nat)) ==>
?P ?nat
```

Inducción asociada a tipos de datos recursivos

- Listas

```
datatype 'a list =
  []
| cons 'a "'a list"

thm list.induct
```

- Esquema de inducción asociado

```
?P [] ==>
(/\x1 x2. ?P x2 ==> ?P (cons x1 x2)) ==>
?P ?list
```

Inducción sugerida por una definición

- Admisión de la función **`sumaImpares`**

```
fun sumaImpares :: "nat => nat" where
  "sumaImpares 0 = 0"
| "sumaImpares (Suc n) = (2*n+1) + sumaImpares n"
```

  - La relación podría ser la descrita por el conjunto $\mathbf{R} = \{(\mathbf{n}, \mathbf{Suc\ n}), \mathbf{n} \in \mathbb{N}\}$
    - La medida que justifica la buena fundamentación es la función identidad
  - Hay una única llamada recursiva: **`sumaImpares n`**
    - La conjetura de terminación es $\mathbf{n} \in \mathbb{N} \rightarrow (\mathbf{n}, \mathbf{Suc\ n}) \in \mathbf{R}$

Inducción sugerida por una definición

- Esquema de inducción asociado a la función **`sumaImpares`**
  - Hay un único caso de inducción con una sustitución
    - $\mathbf{q_1(n)} \equiv \mathbf{(n \in \mathbb{N}^+)}$
    - $\boldsymbol{\sigma_{1,1}} = \mathbf{\{n/n-1\}}$
  - El caso base se caracteriza por $\neg\mathbf{(q_1(n))}$, que restringido a los números naturales es equivalente a $\mathbf{n = 0}$
  - Si $<_{\mathbf{R}}$ es el orden asociado a la relación bien fundamentada **R** y el término $\mathbf{m(n)} \equiv \mathbf{n}$, entonces se tiene

$$\mathbf{q_1(n)} \rightarrow \boldsymbol{\sigma_{1,1}}\mathbf{(m(n))} <_{\mathbf{R}} \mathbf{m(n)}$$

Inducción sugerida por una definición

- Esquema de inducción asociado a la función **`sumaImpares`**
  - Hay un único caso de inducción con una sustitución
    - $\mathbf{q_1(n)} \equiv \mathbf{(n \in \mathbb{N}^+)}$
    - $\boldsymbol{\sigma_{1,1}} = \mathbf{\{n/n-1\}}$
  - El caso base se caracteriza por $\mathbf{\neg(q_1(n))}$, que restringido a los números naturales es equivalente a $\mathbf{n = 0}$
  - Si $<_{\mathbf{R}}$ es el orden asociado a la relación bien fundamentada **R** y el término $\mathbf{m(n)} \equiv \mathbf{n}$, entonces se tiene

$$\mathbf{(n \in \mathbb{N}^+) \rightarrow n - 1 <_R n}$$

  - Que es equivalente a $\mathbf{n \in \mathbb{N} \rightarrow (n, Suc\ n) \in R}$
- La conjetura de terminación que justifica la admisión de la función sirve para justificar el esquema de inducción

Esquemas de inducción en Isabelle

- Definiciones:

```
datatype 'a arbol = Hoja "'a"
                  | Nodo "'a arbol" "'a arbol"

fun numeroHojas :: "'a arbol => nat" where
  "numeroHojas (Hoja x) = 1"
| "numeroHojas (Nodo i d) = numeroHojas i + numeroHojas d"

fun aplana :: "'a arbol => 'a list" where
  "aplana (Hoja x) = [x]"
| "aplana (Nodo i d) = (aplana i)@(aplana d)"
```

Esquemas de inducción en Isabelle

- Conjetura:

```
lemma numeroHojasLengthAplana:
  "numeroHojas a = length (aplana a)"
```

- Posibles inducciones para demostrar la conjetura
  - Inducción asociada al tipo de dato **`arbol`**
  - Inducción sugerida por **`(aplana a)`**
  - Inducción sugerida por **`(numeroHojas a)`**
- Es el mismo esquema de inducción

Esquemas de inducción en Isabelle

- Inducción asociada al tipo de dato: **`(induct a)`**

```
1. /\x. numeroHojas (Hoja x) =
        length (aplana (Hoja x))
2. /\a1 a2.
      numeroHojas a1 = length (aplana a1) ==>
      numeroHojas a2 = length (aplana a2) ==>
      numeroHojas (Nodo a1 a2) =
      length (aplana (Nodo a1 a2))
```

Esquemas de inducción en Isabelle

- Definiciones:

```
fun partePar :: "'a list => 'a list" where
  "partePar []          = []"
| "partePar [x]         = [x]"
| "partePar (x#(y#xs)) = x#(partePar xs)"

fun parteImpar :: "'a list => 'a list" where
  "parteImpar []          = []"
| "parteImpar [x]         = []"
| "parteImpar (x#(y#xs)) = y#(parteImpar xs)"

fun mezclaZip :: "'a list => 'a list => 'a list" where
  "mezclaZip [] ys          = ys"
| "mezclaZip xs []         = xs"
| "mezclaZip (x#xs) (y#ys) = x#(y#(mezclaZip xs ys))"
```

Esquemas de inducción en Isabelle

- Conjetura:

```
lemma mezclaZipPartesParImpar:
  "mezclaZip (partePar xs) (parteImpar xs) = xs"
```

- Posibles inducciones para demostrar la conjetura
  - Inducción asociada al tipo de dato lista
  - Inducción sugerida por **`partePar xs`**
  - Inducción sugerida por **`parteImpar xs`**
    - Igual al sugerido por **`partePar xs`**

Esquemas de inducción en Isabelle

- Inducción asociada al tipo de dato lista: **`(induct xs)`**

```
1. mezclaZip (partePar []) (parteImpar []) = []
2. /\a xs.
      mezclaZip (partePar xs) (parteImpar xs) = xs ==>
      mezclaZip (partePar (a # xs))
                (parteImpar (a # xs)) = a # xs
```

Esquemas de inducción en Isabelle

- Esquema de inducción asociado a la función **partePar**

```
?P [] ==>
(/\x. ?P [x]) ==>
(/\x y xs. ?P xs ==> ?P (x # y # xs)) ==>
?P ?a0.0
```

- Inducción sugerida por **partePar xs**:

```
(induct xs rule: partePar.induct)
```

```
1. mezclaZip (partePar []) (parteImpar []) = []
2. /\x. mezclaZip (partePar [x]) (parteImpar [x]) = [x]
3. /\x y xs.
      mezclaZip (partePar xs) (parteImpar xs) = xs ==>
      mezclaZip (partePar (x # y # xs))
                (parteImpar (x # y # xs)) = x # y # xs
```

Esquemas de inducción en Isabelle

- Definiciones:

```
fun menorLista :: "int => int list => bool" where
  "menorLista a []     = True"
| "menorLista a (x#xs) = (a <= x /\ menorLista a xs)"

fun ordenada :: "int list => bool" where
  "ordenada []     = True"
| "ordenada (x#xs) = (menorLista x xs /\ ordenada xs)"

fun mezcla :: "int list => int list => int list" where
  "mezcla [] ys = ys"
| "mezcla xs [] = xs"
| "mezcla (x#xs) (y#ys) = (if x <= y
                           then x # mezcla xs (y#ys)
                           else y # mezcla (x#xs) ys)"
```

Esquemas de inducción en Isabelle

- Conjetura:

```
lemma ordenada_mezcla:
  assumes "ordenada xs"
          "ordenada ys"
  shows   "ordenada (mezcla xs ys)"
```

- Posibles inducciones para demostrar la conjetura
  - Inducción asociada al tipo de dato lista
  - Inducción sugerida por **`ordenada xs`**
  - Inducción sugerida por **`ordenada ys`**
  - Inducción sugerida por **`mezcla xs ys`**

Esquemas de inducción en Isabelle

- Inducción asociada al tipo de dato lista: **`(induct xs)`**

```
1. ordenada (mezcla [] ys)
2. /\a xs.
      ordenada (mezcla xs ys) ==>
      ordenada (mezcla (a # xs) ys)
```

- Inducción asociada al tipo de dato lista: **`(induct ys)`**

```
1. ordenada (mezcla xs [])
2. /\a ys.
      ordenada (mezcla xs ys) ==>
      ordenada (mezcla xs (a # ys))
```

Esquemas de inducción en Isabelle

- Esquema de inducción asociado a la función **ordenada**

```
?P [] ==>
(/\x xs. ?P xs ==> ?P (x # xs)) ==>
?P ?a0.0
```

- Inducción sugerida por **ordenada xs**:

```
(induct xs rule: ordenada.induct)
```

```
1. ordenada (mezcla [] ys)
2. /\x xs.
      ordenada (mezcla xs ys) ==>
      ordenada (mezcla (x # xs) ys)
```

- El mismo esquema de inducción que el asociado a las listas

Esquemas de inducción en Isabelle

- Esquema de inducción asociado a la función **mezcla**

```
(/\ys. ?P [] ys) ==>
(/\v va. ?P (v # va) []) ==>
(/\x xs y ys.
    (x <= y ==> ?P xs (y # ys)) ==>
    (~(x <= y) ==> ?P (x # xs) ys) ==>
    ?P (x # xs) (y # ys)) ==>
?P ?a0.0 ?a1.0
```

- Inducción sugerida por **mezcla xs ys**:

```
(induct xs ys rule: mezcla.induct)
```

```
1. /\ys. ordenada (mezcla [] ys)
2. /\v va.
      ordenada (mezcla (v # va) [])
3. /\x xs y ys.
      (x <= y ==> ordenada (mezcla xs (y # ys))) ==>
      (~(x <= y) ==> ordenada (mezcla (x # xs) ys)) ==>
      ordenada (mezcla (x # xs) (y # ys))
```

Bibliografía

- A. Krauss *Defining Recursive Functions in Isabelle/HOL* (`https://isabelle.in.tum.de/doc/functions.pdf`)