SWP Assigmnment 2

Alexander Frewein (1430019)
Klaus Fabian Frühwirt (1131522)

Institute of Software Technology
alexander.frewein@student.tugraz.at
fabian.fruehwirth@student.tugraz.at

Beispiel 1

EXP Funktion:

$$\delta\ \underline{rfac} = \underline{if\ eq?(n,0)\ then\ 1\ else\ mult(plus(m,minus(n,1)),\ rfac(m,minus(n,1)))}$$

Basis:

$\omega(n)\ =\ 0$
$I(\delta,\ \omega,\ \underline{rfac(m,n)})$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ eq?(n,0)\ then\ 1\ else\ mult(plus(m,minus(n,1)),\ rfac(m,minus(n,1)))})$

$NR:\ I(\delta,\ \omega,\ \underline{eq?(n,0)}\ =\ eq?(I(\delta,\ \omega,\ \underline{n})),\ I(\delta,\ \omega,\ \underline{0}))$
$eq?(\omega(\underline{n}),\ 0)\ =\ eq?(0,\ 0)$
$=\ 0 ==\ 0\ =\ T$

$I(\delta,\ \omega,\ \underline{1})$
$\rightarrow 1 = m^{\overline{0}}$

Schritt: Hypothese: $(m+n) \stackrel{!}{=} m^{\overline{n}}$

$\omega(n)\ =\ n+1$
$I(\delta,\ \omega,\ \underline{rfac(m,n)})$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ eq?(n,0)\ then\ 1\ else\ mult(plus(m,minus(n,1)),\ rfac(m,minus(n,1)))})$

$NR:\ I(\delta,\ \omega,\ \underline{eq?(n,0)}\ =\ eq?(I(\delta,\ \omega,\ \underline{n})),\ I(\delta,\ \omega,\ \underline{0}))$
$eq?(\omega(\underline{n}),\ 0)\ =\ eq?(n+1,\ 0)$
$\rightarrow n+1 == 0 = F$

$= I(\delta,\ \omega,\ \underline{rfac(m,n))}$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ eq?(n,0)\ then\ 1\ else\ mult(plus(m,minus(n,1)),\ rfac(m,minus(n,1)))})$
$= mult(I(\delta,\ \omega,\ \underline{n}),\ I(\delta,\ \omega,\ \underline{rfac(m,n)}))$

Neue Environment: $\omega(n)\ =\ I(\delta,\ \omega,\ \underline{minus(n,1)})$

$= minus(I(\delta,\ \omega,\ \underline{n}), I(\delta,\ \omega,\ \underline{1}))$
$= minus(\omega(\underline{n}),\ 1)$
$= minus(n+1,1)$
$n+1-1$
$\underline{n}$

$= mult(I(\delta,\ \omega,\ plus(m,minus(n,1)),\ I(\delta,\ \omega',\ rfrac(m,n)))$
$= mult(plus(m,minus(\omega(n),1)), m^{\overline{w'(n)}})$
$= mult(plus(m,n+1-1), m^{\overline{w'(n)}})$
$= mult(m+n), m^{\overline{w'(n)}})$
$\rightarrow (m+n) * m^{\overline{n}} = m^{\overline{n+1}}$

Beispiel 2

EXP Funktion:

$\delta\ \underline{(parity)} = \underline{if\ isEmpty?(xs)\ then\ F\ else\ XOR(first(xs),\ parity(rest(xs))}$
$\delta\ \underline{XOR(p,q)} = \underline{and(or(p,q),\ not(and(p,q)))}$

Basis:

$\omega(xs)\ =\ nil$
$I(\delta,\ \omega,\ \underline{parity(xs))}$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ isEmpty?(xs)\ then\ F\ else\ XOR(first(xs),\ parity(rest(xs))))}$

$NR : I(\delta,\ \omega,\ \underline{isEmpty(xs))}$
$= isEmpty?(\overline{I(\delta,\ \omega,\ \underline{xs})})$
$= isEmpty?(nil) == T$

$\omega(xs)\ =\ nil$
$I(\delta,\ \omega,\ F)$
$= F = parity([\ ])$

Es wird überprüft, ob die Liste leer ist, somit ist die Basis für den Induktionsschritt gegeben.
Schritt:
Hypothese: $parity(xs) \stackrel{!}{=} x0 \oplus x1 \oplus x2 \oplus ..... \oplus xn$
Die parity Funktion ist gemäß Hypothese definiert und im Indokutionsschritt wollen wir Beweisen, dass unsere Funktion diese Hypothese erfüllt.

$\omega(xs) \ = \ x_{n+1} :: xs$
$I(\delta,\ \omega,\ \underline{parity(x_{n+1} :: xs)})$
$= I(\delta,\ \omega,\ \underline{if\ isEmpty?(x_{n+1} :: xs)\ then\ F\ else\ XOR(first(x_{n+1} :: xs),\ parity(rest(x_{n+1} :: xs)))})$

$NR : I(\delta,\ \omega,\ \underline{isEmpty(xs)})$
$= isEmpty?(\overline{I(\delta,\ \omega,\ \underline{xs})})$
$= isEmpty?(x_{n+1} :: xs) == F$

$= I(\delta,\ \omega,\ \underline{XOR(first(x_{n+1} :: xs),\ parity(rest(x_{n+1} :: xs)))})$
$= XOR(first(x_{n+1} :: xs,\ parity(rest(x_{n+1} :: xs)))$
$\rightarrow XOR(x_{n+1}, parity(xs)) == parity(xs) \oplus x_{n+1} \rightarrow q.e.d.$

Mithilfe unseren XOR Funktion haben wir im Schritt bewiesen, dass sie dasselbe Ergebnis liefert, wie die Funktion parity und somit ist unsere EXP Funktion bewiesen.
XOR Funktion: Hypothese: $\oplus = (p \vee q) \ \wedge \ \neg(p \wedge q)$

p	q	$p \vee q$	$p \wedge q$	$\neg p \wedge q$	$(p \vee q) \wedge \neg(p \wedge q)$	$\oplus$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$

Beispiel 3

EXP Funktion:

$\delta\ \underline{sumProd} = \underline{if\ isEmpty?(A)\ then\ z\ else\ plus(mult(first(A), first(B),\ sumProd(rest(A), rest(B)), z)}$

Basis:

$\omega(A) \ = \ nil \quad \omega(B) = nil \quad \omega(z) = z$
$I(\delta,\ \omega,\ \underline{sumProd(A, B, z))}$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ isEmpty?(A)\ then\ z\ else\ plus(mult(first(A), first(B),\ sumProd(rest(A), rest(B)), z)}$

$NR : I(\delta,\ \omega,\ \underline{isEmpty(A)})$
$= isEmpty?(\overline{I(\delta,\ \omega,\ \underline{A})})$
$= isEmpty?(nil) == T$

$= z == sumprod(nil, nil, z)$

Es wird überprüft, ob beide Listen leer ist und ob nur die Konstante z das Ergebnis der Funktion ist, somit ist die Basis für den Induktionsschritt gegeben.
Schritt:
Hypothese: $sumProd(a_{n+1} :: as, b_{n+1} :: bs, z) \stackrel{!}{=} plus(mult(a_{n+1}, b_{n+1}, sumProd(as, bs, z))$

$\omega(A) = a_{n+1} :: as \quad \omega(B) = b_{n+1} :: bs \quad \omega(z) = z$
$I(\delta,\ \omega,\ \underline{sumProd(a_{n+1} :: as, b_{n+1} :: bs, z))}$
$\rightarrow I(\delta,\ \omega,\ \underline{if\ isEmpty?(A)\ then\ z\ else\ plus(mult(first(a_{n+1} :: as), first(b_{n+1} :: bs),\ sumProd(rest(a_{n+1} :: as), rest(b_{n+1} :: bs)), z)}$

$NR : I(\delta,\ \omega,\ \underline{isEmpty(A)})$
$= isEmpty?(\underline{I(\delta,\ \omega,\ A)})$
$= isEmpty?(a_{n+1} :: as) == F$

$I(\delta,\ \omega,\ plus(mult(a_{n+1}, b_{n+1}), I(\delta,\ \omega,\ sumProd(a_s, b_s, z))$
$= plus(I(\delta\ \omega,\ mult(a_{n+1}, b_{n+1}), I(\delta,\ \omega,\ sumProd(a_s, b_s, z)) \rightarrow q.e.d.$

Im Induktionsschritt wird mithilfe der Rekrusion sumProd so oft aufgerufen bis a_s, und b_s leer sind also nil und danach wird das z am Ende dazugezählt. Dies führt dazu, dass sobald die beiden Listen leer sind die Form $sumprod(nil, nil, z)$ entsteht, welche in der Basis definiert ist und dazu führt dass z dazugezählt wird.

Beispiel 4

Scala Funktion:

```
def drop(n : Int, xs: List[Int]): List[Int] = (n, xs) match {
  case (0, xs) => xs
  case (_, Nil) => xs
  case (_, xs::rest) => drop(n-1, rest)
 }
```

Basis:

$drop(0, xs) = xs$

Schritt:
Hypothese: $drop(m, drop(n, xs)) \stackrel{!}{=} drop(m + n, s)$
Fall 1: $|xs| \geqslant m + n$

$drop(m, drop(n, xs)) = drop(m + n, s)$
$= drop(m + 1, drop(n - 1, xs)) = drop(m + n, xs)$
$= drop(m + n, drop(n - n, xs)) = drop(m + n, xs)$
$= drop(m + n, drop(0, xs)) = drop(m + n, xs)$
$\rightarrow drop(0, xs) = xs\ [lautBasis]$
$= drop(m + n, xs) = drop(m + n, xs) \quad \rightarrow q.e.d.$

In diesem Fall wird bewiesen das beide Ausdrücke equivalent sind und ob die Anzahl der Elemente gleich sind wie jene welche entfernt wurden. **Fall 2:** $|xs| < m + n$

$drop(m, drop(n, xs)) = drop(m + n, s)$
$= drop(m + 1, drop(n - 1, xs)) = drop(m + n, xs)$
$= drop(m + n, nil) = drop(m + n, nil) \quad [def.drop]$
$\rightarrow q.e.d.$

Da in Fall 1 bereits bewiesen wurde das beide Ausdrücke equivalent sind, wird in diesem Fall nur noch behandelt wenn die Anzahl der Elemente in einer Liste kleiner sind als die Anzahl der Elemente welche entfernt werden. Das Ergebnis ist, dass dadurch eine Leere Liste per Programm zurückgegeben wird.

Beispiel 5

Scala Funktion:

```
def width(t: Tree): Int = t match {
  case t: Branch => width(t.left) + width(t.right)
  case t: Leaf => 1
}

def depth(t: Tree): Int = t match {
  case t: Branch => math.max((1+depth(t.left)), (1+depth(t.right)))
  case _: Leaf => 0
}
```

Beispiel 6

Scala Funktion:

```
def sumProd(xs: List[Int], ys: List[Int], z: Int): Int =
  (xs, ys) match {
    case (x::xs, y::ys) =>  sumProd(xs, ys, z + (x * y))
    case (Nil, _) => z
}
```

Basis:

$sumProd(Nil, Nil, z) \stackrel{!}{=} sum(zipMul(Nil, Nil), z)$

$z = sum(Nil, z)$

$z = z$

Schritt:
Hypothese: $sumProd(x :: xs, y :: ys, z) \stackrel{!}{=} sum(zipMul(x :: xs, y :: ys), z)$

$sumProd(xs, ys, (x * y) + z) = sum((x * y) :: zipMul(xs, ys), z)$
$sumProd(xs, ys, z + (x * y)) = sum(zipMul(xs, ys), z + (x * y))$
$sumProd(xs, ys, z) + (x * y) = sum(zipMul(xs, ys), z) + (x * y) \quad q.e.d.$

Die Funktion sumProd addiert und errechnet das Skalarprodukt innerhalb der gleichen während sum und zipMul dies aufteilt. Aufgrund der letzen Umformung ist zusehen, dass bei beiden das gleiche Ergebnis liefert und somit sumProd immer das gleiche Ergebnis liefert wie sum und zipMul zusammen.