Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

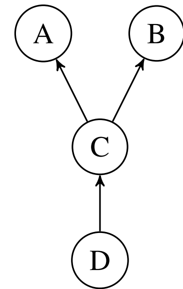
RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 12: Grundlagen der Programmierung (WS 2024/25)

Aufgabe 1 Untertypen (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit Untertypbeziehungen beschäftigen. Sie können sich an den Vorlesungsfolien 901 bis 969 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

Es gelten die Untertypbeziehungen $D \preccurlyeq C$, $C \preccurlyeq A$ und $C \preccurlyeq B$, die in der nebenstehenden Abbildung visualisiert sind.



In der folgenden Tabelle werden je zwei Typen in Relation gesetzt:

- $t_1 \prec t_2$ bedeutet, dass t_1 ein Untertyp von t_2 ist ($t_1 \preccurlyeq t_2$), nicht jedoch t_2 ein Untertyp von t_1.
- $t_1 \succ t_2$ bedeutet, dass t_2 ein Untertyp von t_1 ist ($t_2 \preccurlyeq t_1$), nicht jedoch t_1 ein Untertyp von t_2.
- $t_1 \parallel t_2$ bedeutet, dass t_1 und t_2 unvergleichbar sind, das heißt es gilt weder $t_1 \preccurlyeq t_2$ noch $t_2 \preccurlyeq t_1$.

Füllen Sie die Lücken in der Tabelle aus. In der ersten und dritten Spalte müssen Sie einen Typ eintragen, in der zweiten Spalte eine Relation ($\prec$ oder $\succ$ oder $\parallel$).

Zur Erinnerung: Die folgenden Deduktionsregeln gelten für die Relation »$\preccurlyeq$«:

$$\frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3} \qquad \frac{t_1 \preccurlyeq t_1' \qquad t_2 \preccurlyeq t_2'}{t_1 * t_2 \preccurlyeq t_1' * t_2'} \qquad \frac{t_1' \preccurlyeq t_1 \qquad t_2 \preccurlyeq t_2'}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'}$$

$\mathbf{t_1}$	**Relation**	$\mathbf{t_2}$
D	$\boxed{\prec}$	A
$\boxed{A}$ oder $\boxed{B}$	$\succ$	C
$A * C$	$\prec$	$\boxed{A * A}$ oder $\boxed{A * B}$
$A * B$	$\boxed{\parallel}$	$B * A$
$C * D$	$\succ$	$\boxed{D * D}$
$A \to C$	$\boxed{\prec}$	$C \to C$
$B \to D$	$\boxed{\prec}$	$C \to C$
$C \to D$	$\succ$	$\boxed{A \to D}$ oder $\boxed{B \to D}$
$C \to A$	$\boxed{\succ}$	$B \to D$
$D \to A$	$\boxed{\parallel}$	$C \to B$

Aufgabe 2 Mini CAS (Einreichaufgabe, 30 Punkte)

Motivation: In dieser Aufgabe sollen Sie ein Problem sowohl funktional als auch objektorientiert implementieren und beide Modelle miteinander vergleichen.

Schreiben Sie Ihre Lösungen in die Datei `Calculus.fs` *aus der Vorlage* `Aufgabe-12-2.zip`.

Bei ihrem Job als Mathe-Nachhilfelehrerin muss Lisa Lista viele Aufgaben korrigieren. Um sich die Arbeit etwas zu erleichtern, beschließt sie ein kleines Computer Algebra System (CAS) zu entwickeln, mit dem sie Ableitungen von einigen eindimensionalen Funktionen symbolisch bestimmen kann.

Lisa Lista schwebt folgende funktionale Modellierung vor:

```
type Function =
    | Constant of Nat                 // Konstante Funktion
    | Id                              // Identität
    | Add of (Function * Function)    // Addition
    | Mul of (Function * Function)    // Multiplikation
    | Pow of (Function * Nat)         // Potenz
    | Comp of (Function * Function) // Verkettung
```

Die Idee ist, dass jedes Element vom Typ `Function` eine reelle Funktion in unserem CAS repräsentiert, die wir jedoch nur an den Stellen der natürlichen Zahlen ($\mathbb{R} \cap \mathbb{N}$) auswerten (damit sind Ableitungen wie gewohnt definiert und wir vermeiden es in F# Fließkommazahlen zu verwenden).

So repräsentiert zum Beispiel `Constant 1N` die Funktion $f(x) = 1$, `Id` repräsentiert $f(x) = x$, `Add (Id, Constant 2N)` repräsentiert $f(x) = x + 2$, `Mul (Id, Constant 2N)` repräsentiert $f(x) = x \cdot 2$ und `Add (Mul(Id, Id), Mul(Constant 2N, Id))` repräsentiert $f(x) = x \cdot x + 2 \cdot x$.

Außerdem können wir Funktionen mit `Comp` verketten, zum Beispiel repräsentiert `Mul(Comp(Pow(Id, 2N), Add(Id, Constant 3N)), Add(Id, Constant 4N))` die Funktion $f(x) = (x + 3)^2 \cdot (x + 4)$. Das erste Argument bezeichnet also die äußere Funktion, das zweite Argument die innere Funktion.

Lisa Lista möchte für ihr CAS drei Operationen (F#-Funktionen) implementieren:

- `toString: Function -> String` stellt die gegebene CAS-Funktion (korrekt geklammert) als String dar.
- `apply: Function -> Nat -> Nat` nimmt eine CAS-Funktion `f` sowie eine natürliche Zahl `n` und berechnet das Ergebnis von `f` angewendet auf `n`.
- `derive: Function -> Function` berechnet die Ableitung der gegebenen CAS-Funktion.

Harry Hacker schlägt eine objektorientierte Modellierung vor und präsentiert folgende Schnittstelle:

```
type IFunction =
    interface
        abstract member ToString: Unit -> String
        abstract member Apply: Nat -> Nat
        abstract member Derive: Unit -> IFunction
    end
```

Er möchte dann für jede Art von unterstützter CAS-Funktion einen Objektkonstruktor implementieren, also eine F#-Funktion die ein Objekt vom Typ `IFunction` zurückgibt.

Lisa Lista und Harry Hacker haben bereits begonnen, das CAS zu implementieren. Sie finden den unfertigen Code im Template zu dieser Aufgabe. Beide haben die Operationen zur Darstellung als String und zum Anwenden der Funktion für die konstante Funktion, die Identität und die Addition bereits umgesetzt. Es fehlen noch Multiplikation, Potenz und Verkettung sowie die Berechnung der Ableitung.

Hinweis: Zu dieser Aufgabe gibt es Testfälle im ExClaim-System, die jedoch erst dann kompilieren, wenn Teilaufgabe a und b bearbeitet wurden.

a) Erweitern Sie beide Modelle um Multiplikation, Potenz und Verkettung. Für Lisas funktionales Modell müssen Sie dazu zunächst in der Typdefinition die drei vorbereiteten Zeilen aktivieren.

Hinweise: Stellen Sie die Potenzfunktion mit dem Zeichen ^ dar, die Funktion `Pow (Id, 2N)` demnach als `x^2`. Zur Implementierung von `apply` können Sie den F#-Operator `**` verwenden. Zur Darstellung der Funktionskomposition wird üblicherweise das Zeichen ∘ verwendet, nutzen Sie der Einfachheit halber den Kleinbuchstaben o.

Funktionales Modell

Zusätzliche Fälle für toString:

```
| Mul (f1, f2)  -> "(" + toString f1 + " * " + toString f2 + ")"
| Pow (f1, n)   -> toString f1 + " ^ " + show n
| Comp (f1, f2) -> "(" + toString f1 + " o " + toString f2 + ")"
```

Zusätzliche Fälle für apply:

```
| Mul (f1, f2)  -> apply f1 x * apply f2 x
| Pow (f1, n)   -> apply f1 x ** n
| Comp (f1, f2) -> apply f1 (apply f2 x)
```

Objektorientiertes Modell (die `Derive` members sind bereits für die nächste Teilaufgabe)

```
let rec mul (f1: IFunction, f2: IFunction): IFunction =
    { new IFunction with
        member self.ToString (): String =
            "(" + f1.ToString () + " * " + f2.ToString () + ")"
        member self.Apply (x: Nat): Nat =
            f1.Apply(x) * f2.Apply(x)
        member self.Derive (): IFunction =
            add (mul (f1.Derive(), f2),  mul (f1, f2.Derive()))
    }

let rec pow (f1: IFunction, n: Nat): IFunction =
    { new IFunction with
        member self.ToString (): String =
            f1.ToString () + " ^ " + show n
        member self.Apply (x: Nat): Nat =
            f1.Apply(x) ** n
        member self.Derive (): IFunction =
            mul (mul (constant n, pow (f1, (n-1N))), f1.Derive())
    }

let rec comp (f1: IFunction, f2: IFunction): IFunction =
    { new IFunction with
        member self.ToString (): String =
            "(" + f1.ToString () + " o " + f2.ToString () + ")"
        member self.Apply (x: Nat): Nat =
            f1.Apply(f2.Apply(x))
        member self.Derive (): IFunction =
            mul (comp (f1.Derive(), f2), f2.Derive())
    }
```

b) Erweitern Sie beide Modelle um die Berechnung der Ableitung. Für Harrys objektorientiertes Modell müssen Sie dazu zunächst in der Schnittstellendefinition die vorbereitete Zeile aktivieren.

Hinweise: Für die Ableitungsregeln können Sie Ihr altes Mathematik-Schulbuch oder eine Formelsammlung konsultieren. Denken Sie an die Produktregel und die Kettenregel!

Funktionales Modell

```
let rec derive (f: Function): Function =
    match f with
    | Constant n     -> Constant 0N
    | Id             -> Constant 1N
    | Add (f1, f2)   -> Add (derive f1, derive f2)
    | Mul (f1, f2)   -> Add (Mul (derive f1, f2), Mul (f1, derive f2))
    | Pow (f1, n)    -> Mul (Mul (Constant n, Pow (f1, n - 1N)), derive f1)
    | Comp (f1, f2) -> Mul (Comp (derive f1, f2), derive f2)
```

Objektorientiertes Modell (siehe auch Lösung der vorherigen Teilaufgabe)

Erweiterung von `const`

```
member self.Derive (): IFunction = constant 0N
```

Erweiterung von `id`

```
member self.Derive (): IFunction = constant 1N
```

Erweiterung von `add`

```
member self.Derive (): IFunction = add (f1.Derive(), f2.Derive())
```

c) Nachdem Sie sowohl das funktionale als auch das objektorientierte Modell erweitert haben, beantworten Sie dazu folgende Fragen. Begründen Sie Ihre Antworten (jeweils maximal 50 Wörter).

1. Welches der Modelle lässt sich leichter um zusätzliche Operationen (apply, toString, derive) erweitern?

   Das funktionale Modell, weil man hier lediglich neue F#-Funktionen hinzufügen muss und keine Änderung der bisherigen Definitionen notwendig ist.

2. Welches der Modelle lässt sich leichter um zusätzliche Arten von CAS-Funktionen (constant, id, add, mul, pow, comp) erweitern?

   Das objektorientierte Modell, weil man hier lediglich neue Objektkonstruktoren hinzufügen muss und keine Änderung der bisherigen Definitionen notwendig ist.

3. Welches der Modelle ist im Bezug auf künftige Erweiterungen unseres CAS die bessere Wahl?

   Es ist wahrscheinlicher, dass zusätzliche Arten von Funktionen ergänzt werden (denkbar wären beispielsweise trigonometrische Funktionen wie Sinus und Cosinus), daher ist das objektorientierte Modell die bessere Wahl.

Aufgabe 3 Untertypen (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie die Regeln der statischen Semantik von Schnittstellen und Untertypen einüben. Sie können sich an den Vorlesungsfolien 899 bis 967 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

```
type A =
    interface
        abstract member f: Unit -> Nat
    end

type B =
    interface
        inherit A
        abstract member g: Nat -> String
    end

type C =
    interface
        inherit A
        abstract member h: String -> Nat
    end

type D =
    interface
        inherit C
        abstract member i: Nat -> Unit
    end
```

Verwenden Sie die Schnittstellentypdefinitionen von oben, um den Typ der folgenden Ausdrücke mit einem vollständigen Beweisbaum anzugeben. Benutzen Sie die Regeln der **statischen Semantik** aus der Vorlesung.

a) **fun** `(s : B * D) -> ((snd s) :> C).h ((fst s).g 1N)`

Verwende $\Sigma := \{\texttt{s} \mapsto B * D\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : B * D}}{\Sigma \vdash \texttt{snd s} : D} \quad \dfrac{}{D \preccurlyeq C}}{\Sigma \vdash \texttt{(snd s) :> C} : C}}{\Sigma \vdash \texttt{((snd s) :> C).h} : String \to Nat} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : B * D}}{\Sigma \vdash \texttt{fst s} : B}}{\Sigma \vdash \texttt{(fst s).g} : Nat \to String} \quad \dfrac{}{\Sigma \vdash \texttt{1N : Nat}}}{\Sigma \vdash \texttt{(fst s).g 1N} : String}}{\Sigma \vdash \texttt{((snd s) :> C).h ((fst s).g 1N)} : Nat}}{\emptyset \vdash \texttt{fun (s : B * D) -> ((snd s) :> C).h ((fst s).g 1N)} : B * D \to Nat}$$

b) **fun** `(s: A -> D) -> (`**fun** `(b: B) -> (s b).f ())`

Verwende $\Sigma := \{\texttt{s} \mapsto A \to D, b \mapsto B\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : A \to D} \quad \dfrac{\dfrac{}{B \preccurlyeq A} \quad \dfrac{\dfrac{}{D \preccurlyeq C} \quad \dfrac{}{C \preccurlyeq A}}{D \preccurlyeq A}}{\Sigma \vdash A \to D \preccurlyeq B \to A}}{\Sigma \vdash \texttt{s} : B \to A} \quad \dfrac{}{\Sigma \vdash \texttt{b} : B}}{\Sigma \vdash \texttt{s b} : A}}{\Sigma \vdash \texttt{(s b).f} : Unit \to Nat} \quad \dfrac{}{\Sigma \vdash \texttt{()} : Unit}}{\Sigma \vdash \texttt{(s b).f ()} : Nat}}{\{\texttt{s} \mapsto A \to D\} \vdash \texttt{fun (b: B) -> (s b).f ()} : B \to Nat}}{\emptyset \vdash \texttt{fun (s: A -> D) -> (fun (b: B) -> (s b).f ())} : (A \to D) \to (B \to Nat)}$$