40. Knobelaufgabe #24

Die Queen feiert in Kürze ihren 100. Geburtstag. Aus Anlass des runden Jubiläums plant sie, ihren gutsortierten Weinkeller zu öffnen und die edlen Tropfen ihren Gästen zu kredenzen.

Drei Wochen vor den Feierlichkeiten teilt ihr der Geheimdienst mit, dass eine ihrer rund tausend Weinflaschen vergiftet wurde — man wisse, aber nicht welche. Aber man kenne das Gift: Ein winziger Schluck führt zum Tode; allerdings nicht unmittelbar, sondern mit einer Verzögerung von zwei Wochen.

Die Queen ist bereit, Opfer zu bringen und einige ihrer Diener zum Probetrinken abzustellen. Für die Vorbereitung der Feierlichkeiten wird aber eigentlich das gesamte Personal benötigt. Wieviele Diener muss die Queen höchstens abstellen, um die vergiftete Weinflasche zu identifizieren? Tausend Diener reichen sicherlich aus, aber kommt sie auch mit weniger aus?

40. Motivation

Zusätzliche Funktionalität: Kontoauszug.

```
type IAccountPlus =
  interface
    inherit IAccount
    abstract member Statement : Array ⟨Nat⟩
  end
```

☞ *IAccountPlus* bietet zusätzlich die Möglichkeit an, einen Kontoauszug anzufordern.

Lies: die Schnittstelle *IAccountPlus erweitert* die Schnittstelle *IAccount*. (Das Schlüsselwort ***inherit*** ist etwas unglücklich gewählt.)

40. Motivation

Das Objekt *ludwigs* lässt sich zu einem Plus-Konto erweitern.

$$
\begin{array}{l}
\textbf{let}\ \mathit{ludwigs} : \mathit{IAccountPlus} = \\
\quad \textbf{let}\ \mathit{size} \quad\quad = 10 \\
\quad \textbf{let}\ \mathit{history} \quad = [\,|\ \textbf{for}\ k\ \textbf{in}\ 0\,..\,\mathit{size}-1 \rightarrow 0\ |\,] \\
\quad \textbf{let mutable}\ i = 0 \\
\quad \textbf{let}\ \mathit{next}\ () \quad = (i+1)\ \%\ \mathit{size} \\
\quad \{ \\
\quad\quad \textbf{new}\ \mathit{IAccountPlus}\ \textbf{with} \\
\quad\quad\quad \textbf{member}\ \mathit{self}.\mathit{Deposit}\ (\mathit{amount} : \mathit{Nat}) = \\
\quad\quad\quad\quad \mathit{history}.[\mathit{next}\ ()] \leftarrow \mathit{history}.[i] + \mathit{amount};\ i \leftarrow \mathit{next}\ () \\
\quad\quad\quad \textbf{member}\ \mathit{self}.\mathit{Withdraw}\ (\mathit{amount} : \mathit{Nat}) = \\
\quad\quad\quad\quad \mathit{history}.[\mathit{next}\ ()] \leftarrow \mathit{monus}\ (\mathit{history}.[i], \mathit{amount});\ i \leftarrow \mathit{next}\ () \\
\quad\quad\quad \textbf{member}\ \mathit{self}.\mathit{Balance} = \\
\quad\quad\quad\quad \mathit{history}.[i] \\
\quad\quad\quad \textbf{member}\ \mathit{self}.\mathit{Statement} = \\
\quad\quad\quad\quad [\,|\ \textbf{for}\ k\ \textbf{in}\ 0\,..\,\mathit{size}-1 \rightarrow \mathit{history}.[(\mathit{size}+i-k)\ \%\ \mathit{size}]\ |\,] \\
\quad \}
\end{array}
$$

40. Motivation

Beobachtung: Das Objekt *ludwigs* kann nicht an die Funktion *transfer* übergeben werden!

$transfer : IAccount * Nat * IAccount \to Unit$

$ludwigs : IAccountPlus$

☞ Der Typ von *IAccountPlus* ist nicht *gleich* dem Typ *IAccount*.

40. Motivation

Beobachtung: Der Aufruf

$$\mathit{transfer}\ (\mathit{lisas}, 1000, \mathit{ludwigs})$$

kann aber problemlos ausgerechnet werden, da *ludwigs* mehr Funktionalität als gefordert anbietet — *ludwigs* ist sozusagen überqualifiziert.

☞ Das Typsystem von Mini–F# kann mit Hilfe von *Untertypen* flexibler gemacht werden.

40. Motivation

Idee: ein Element eines Untertyps kann überall da verwendet werden, wo ein Element eines Obertyps verlangt wird. Wir führen eine Untertypbeziehung auf Typen ein.

Zum Beispiel: die Schnittstelle *IAccountPlus* erweitert die Schnittstelle *IAccount* (Schlüsselwort ***inherit***).

$$\mathit{IAccountPlus} \preccurlyeq \mathit{IAccount}$$

Mit Hilfe einer sogenannten Typanpassung (engl. cast) lässt sich ein Element eines Untertyps in ein Element eines Obertyps überführen.

$$\mathit{transfer}\ (\mathit{lisas}, 1000, \mathit{ludwigs} :> \mathit{IAccount})$$

40. Motivation

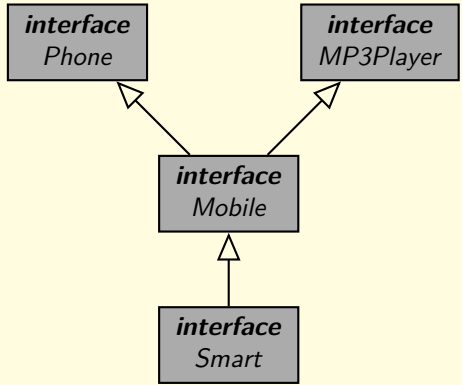
Mehrfachvererbung ist möglich:

```
type Phone =
  abstract member Dial   : Nat → Unit
type MP3Player =
  abstract member Play   : String → Unit
type Mobile =
  inherit Phone
  inherit MP3Player
  abstract member Lookup : String → Nat
  abstract member Call   : String → Unit
type Smart =
  inherit Mobile
  abstract member Browse : String → Unit
```

(Die ***interface*** ... ***end*** Klammer kann weggelassen werden.)

40. Motivation

Schnittstellen als Diagramm:



☞ Bestandteil der „Unified Modeling Language“ (UML), einer Modellierungssprache für Software.

40. Abstrakte Syntax

Deklarationen:

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $U =$	erweiterte Schnittstellentypdefinition
interface	
inherit T_1	
inherit T_2	
abstract member $\ell : t$	
end	

☞ Es dürfen beliebig viele ***inherit*** Klauseln aufgeführt werden und beliebig viele neue Mitglieder hinzukommen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ $e :> t$	Typumwandlung$\diagdown$Typeinschränkung

40. Statische Semantik

Typregel:

$$\frac{\Sigma \vdash e : t' \qquad t' \preccurlyeq t}{\Sigma \vdash (e :> t) : t}$$

☞ Informationsverlust: das Verhalten wird eingeschränkt. (Ein Objekt ist die Summe seines Verhaltens.)

40. Statische Semantik

Gute Nachricht: Typanpassungen werden auch automatisch vorgenommen (engl. automatic upcast).

$$\frac{\Sigma \vdash e : t \qquad t \preccurlyeq \tau'}{\Sigma \vdash e : \tau'}$$

Schlechte Nachricht: die automatische Typanpassung gelingt nicht immer. Problemzonen: Zweige einer Alternative.

☞ Die Relation '$\preccurlyeq$' muss mit Leben gefüllt werden.

40. Untertypen: Quasiordnung

Die Untertypbeziehung ist *reflexiv* und *transitiv*.

$$\frac{}{t \preccurlyeq t}$$

$$\frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3}$$

☞ '$\preccurlyeq$' ist eine sogenannte *Quasiordnung*.

40. Untertypen: *inherit*

Eine Typdefinition mit ***inherit*** Klauseln

type $U =$
  interface
    inherit T_1
    inherit T_2
    abstract member $\ell : t$
  end

legt *explizit* die folgenden Beziehungen fest (Axiome):

$$\frac{}{U \preccurlyeq T_1} \qquad \frac{}{U \preccurlyeq T_2}$$

40. Untertypen: Paare

Wie ist '$\preccurlyeq$' auf strukturierten Typen wie zum Beispiel dem Paartyp $t_1 * t_2$ definiert?

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

☞ Warum ist das eine sinnvolle Festlegung?

40. Untertypen: Paare

Welche Operationen werden auf Paare angewendet? Die Projektionsfunktionen *fst* und *snd*.

Die Untertypregel für Paare lässt sich herleiten, wenn man die Typregeln für *fst* und *snd* studiert.

$$\frac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{fst}\ e : t_1} \qquad \dfrac{}{t_1 \preccurlyeq t_1'}}{\Sigma \vdash \mathit{fst}\ e : t_1'} \qquad \frac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2}}{\Sigma \vdash \mathit{snd}\ e : t_2} \qquad \dfrac{}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash \mathit{snd}\ e : t_2'}$$

Die Paarregel erlaubt es alternativ, direkt das Paar anzupassen:

$$\frac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\Sigma \vdash e : t_1' * t_2'}}{\Sigma \vdash \mathit{fst}\ e : t_1'} \qquad \frac{\dfrac{\dfrac{\dots}{\Sigma \vdash e : t_1 * t_2} \qquad \dfrac{}{t_1 * t_2 \preccurlyeq t_1' * t_2'}}{\Sigma \vdash e : t_1' * t_2'}}{\Sigma \vdash \mathit{snd}\ e : t_2'}$$

40. Untertypen: Funktionen

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

Der Funktionstyp ist

- *kontravariant* im Argumenttyp ($t_1' \preccurlyeq t_1$) und
- *kovariant* im Ergebnistyp ($t_2 \preccurlyeq t_2'$).

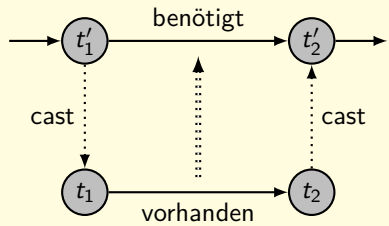
40. Untertypen: Funktionen

Die Varianz des Funktionstyps lässt sich herleiten, wenn man die Typregel der Funktionsapplikation studiert.

$$\dfrac{\dfrac{}{\Sigma \vdash e : t_1 \to t_2} \qquad \dfrac{\dfrac{}{\Sigma \vdash e_1 : t_1'} \qquad \dfrac{}{t_1' \preccurlyeq t_1}}{\Sigma \vdash e_1 : t_1}}{\dfrac{\Sigma \vdash e\ e_1 : t_2 \qquad \dfrac{}{t_2 \preccurlyeq t_2'}}{\Sigma \vdash e\ e_1 : t_2'}}$$

40. Untertypen: Funktionen

Das folgende Diagramm illustriert die Typanpassungen: Eine Funktion des Typs $t_1' \to t_2'$ kann über den Umweg $t_1 \to t_2$ konstruiert werden.



Die Richtung der Typanpassungen kehrt sich für Argument und Ergebnis um.

40. Untertypen: Funktionen – Beispiele

$$
\begin{aligned}
\mathit{Smart} \to \mathit{Smart} &\preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
\mathit{Phone} \to \mathit{Smart} &\preccurlyeq \mathit{Smart} \to \mathit{Smart} \\
\mathit{Phone} \to \mathit{Smart} &\preccurlyeq \mathit{Smart} \to \mathit{Phone} \\
(\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart} &\preccurlyeq (\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone} \\
((\mathit{Phone} \to \mathit{Smart}) \to \mathit{Phone}) \to \mathit{Smart} &\preccurlyeq \\
((\mathit{Smart} \to \mathit{Phone}) \to \mathit{Smart}) \to \mathit{Phone}&
\end{aligned}
$$

☞ In dem Typ $((t_1 \to t_2) \to t_3) \to t_4$ stehen t_1 und t_3 an kontravarianten und t_2 und t_4 an kovarianten Positionen.

Die Richtung der Quasiordnung wechselt mit jeder Schachtelung im Argumenttyp.

40. Untertypen: Speicherzellen

Eine Speicherzelle kann mit ‘ !’ gelesen und mit ‘:=’ geschrieben werden.
Der Lesezugriff ist kovariant, der Schreibzugriff ist kontravariant.

Da Speicherzellen beide Zugriffsarten unterstützen, sind sie summa summarum *invariant*.

$$\frac{t' \preccurlyeq t \qquad t \preccurlyeq t'}{\mathit{Ref}\langle t' \rangle \preccurlyeq \mathit{Ref}\langle t \rangle}$$

☞ Entsprechendes gilt für alle modifizierbaren Datenstrukturen (engl. mutable data structures) wie zum Beispiel Arrays.

40. Untertypen: Speicherzellen

Die Invarianz von *Ref* lässt sich herleiten, wenn man die Typregeln für die Dereferenzierung ' !' und die Zuweisung ':=' studiert.

$$\frac{\dfrac{\overline{\Sigma \vdash e : \mathit{Ref}\langle t \rangle}}{\Sigma \vdash\ !e : t} \qquad \overline{t \preccurlyeq t'}}{\Sigma \vdash\ !e : t'}$$

$$\frac{\overline{\Sigma \vdash e_1 : \mathit{Ref}\langle t \rangle} \qquad \dfrac{\overline{\Sigma \vdash e_2 : t'} \qquad \overline{t' \preccurlyeq t}}{\Sigma \vdash e_2 : t}}{\Sigma \vdash (e_1 := e_2) : \mathit{Unit}}$$

40. Untertypen: Speicherzellen — da capo⋆

☞ Unterschiede man mit Hilfe des Typsystems zwischen lesendem und schreibendem Zugriff, ergäbe sich ein verfeinertes Bild.

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash \textbf{\textit{ref}}\ e : \mathit{Ref}\langle t\rangle}$$

$$\frac{\Sigma \vdash e : \mathit{Read\text{-}Ref}\langle t\rangle}{\Sigma \vdash\ !e : t}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Write\text{-}Ref}\langle t\rangle \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : \mathit{Unit}}$$

☞ *Read-Ref* ist der Typ einer 'read-only' Speicherzelle; *Write-Ref* ist der Typ einer 'write-only' Speicherzelle.

40. Untertypen: Speicherzellen — da capo⋆

Der Typ *Read-Ref* ist kovariant, *Write-Ref* hingegen kontravariant.

$$\frac{t' \preccurlyeq t}{\mathit{Read\text{-}Ref}\langle t'\rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t\rangle} \qquad \frac{t \preccurlyeq t'}{\mathit{Write\text{-}Ref}\langle t'\rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t\rangle}$$

Der Typ *Ref* ist der Durchschnitt von *Read-Ref* und *Write-Ref*.

$$\frac{}{\mathit{Ref}\langle t'\rangle \preccurlyeq \mathit{Read\text{-}Ref}\langle t\rangle} \qquad \frac{}{\mathit{Ref}\langle t'\rangle \preccurlyeq \mathit{Write\text{-}Ref}\langle t\rangle}$$

☞ Eine les- und schreibbare Speicherzelle lässt sich zu einer ‘read-only’ oder einer ‘write-only’ Speicherzelle herabstufen.

☞ Wozu sind ‘write-only’ Speicherzellen gut?

40. Dynamische Semantik

Die Auswertung von Programmen ändert sich nicht.

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash (e :> t) \Downarrow \nu}$$

☞ Eine Typanpassung könnte prinzipiell mit einer Umwandlung des Wertes einhergehen.
Zum Beispiel:

- $\mathit{Nat} \preccurlyeq \mathit{Int}$: eine natürliche Zahl wird in der Regel anders repräsentiert als eine ganze Zahl (ohne und mit Vorzeichen);
- $\mathit{Ref}\langle t \rangle \preccurlyeq t$: Speicherzellen werden automatisch dereferenziert.

☞ Sämtliche Auswertungsregeln müssten dann angepasst werden!

40. Vertiefung

☞ Untertypen sind eine Spielart der *Polymorphie*.

Zur Erinnerung: Der Name Polymorphie kommt aus dem Griechischen ($\pi o \lambda \upsilon \mu o \rho \phi \iota \alpha$) und bedeutet Vielgestaltigkeit. Im Kontext von Programmiersprachen meint Polymorphie, dass ein Wert in unterschiedlichen Typkontexten verwendet werden kann. *Kurz:* ein Wert hat mehrere Typen.

Man unterscheidet zwischen 4 verschiedenen Arten von Polymorphie:

	universelle Polymorphie	ad-hoc Polymorphie
konjunktive Polymorphie	parametrische Polymorphie ($\forall$) (engl. generics) *length*	Überladung ($\wedge$) (engl. overloading) $=$, $+$
Inklusions-polymorphie	Untertypen ($\preccurlyeq$) (engl. subtyping) *transfer*	Konversion (engl. coercion) —

40. Parametrische Polymorphie

Die polymorphe Funktion $\mathit{length} : \mathit{List}\,\langle 'a \rangle \to \mathit{Nat}$ kann zum Beispiel die Länge einer Liste eines beliebigen Typs bestimmen; sie besitzt im Prinzip unendliche viele Typen.

$$
\begin{aligned}
&\mathit{length} : \mathit{List}\,\langle \mathit{Nat} \rangle \to \mathit{Nat} \\
&\mathit{length} : \mathit{List}\,\langle \mathit{Bool} \rangle \to \mathit{Nat} \\
&\mathit{length} : \mathit{List}\,\langle \mathit{List}\,\langle \mathit{Nat} \rangle \rangle \to \mathit{Nat} \\
&\mathit{length} : \mathit{List}\,\langle \mathit{List}\,\langle \mathit{Bool} \rangle \rangle \to \mathit{Nat} \\
&\ldots
\end{aligned}
$$

Da *length* einen heimlichen Typparameter hat, spricht man auch genauer von einer *parametrisch polymorphen Funktion* oder etwas weniger sperrig von einer *generischen Funktion* (engl. generic function).

40. Überladung

Von *Überladung* (engl. overloading) spricht man, wenn der *gleiche* Bezeichner für *unterschiedliche* Funktionen für unterschiedliche Werte verwendet wird.

$$
\begin{aligned}
(+) &: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
(+) &: \mathit{float} \to \mathit{float} \to \mathit{float} \\
(+) &: \mathit{String} \to \mathit{String} \to \mathit{String}
\end{aligned}
$$

Überladung ist die kleine Schwester der parametrischen Polymorphie.

40. Überladung

Methodennamen dürfen überladen werden:

```
type IStack ⟨'elem⟩ =
  interface
    abstract member Push : 'elem → Unit
    abstract member Push : List ⟨'elem⟩ → Unit
    abstract member Pop  : Unit → 'elem
    abstract member Top  : 'elem
  end
```

☞ Mit *Push* kann sowohl ein einzelnes Element als auch alle Elemente einer Liste auf einen Stack abgelegt werden.

☞ Um Mehrdeutigkeiten zu vermeiden, müssen sich die *Argumenttypen* der überladenen Methoden unterscheiden.

40. Untertypen

Die Funktion *transfer* ist ebenfalls polymorph; sie kann auf Elemente eines beliebigen Untertyps von *IAccount* angewendet werden.

$$
\begin{aligned}
&\mathit{transfer} : \mathit{IAccount} * \mathit{Nat} * \mathit{IAccount} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccount} * \mathit{Nat} * \mathit{IAccountPlus} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccountPlus} * \mathit{Nat} * \mathit{IAccount} && \to \mathit{Unit}\\
&\mathit{transfer} : \mathit{IAccountPlus} * \mathit{Nat} * \mathit{IAccountPlus} && \to \mathit{Unit}\\
&\ldots
\end{aligned}
$$

Die Zahl der Untertypen von *IAccount* ist zwar endlich, aber unbegrenzt, da die Schnittstellenhierarchie zu jedem Zeitpunkt beliebig erweitert werden kann.

40. Konversion

Von *Konversion* (engl. coercion) spricht man, wenn ein Element eines Typs automatisch in ein Element eines anderen Typs umgewandelt wird. Wird *nicht* unterstützt.

☞ Konversionen müssen *explizit* durchgeführt werden:

```
Mini⟩ float32 123456789
val it : float32 = 123456792.0f
Mini⟩ int it
val it : int = 123456792
```

Weitere böse Überraschungen:

```
Mini⟩ int 1e10f
val it : int = -2147483648
Mini⟩ float32 it
val it : float32 = -2147483650f
```

Konversion ist die kleine Schwester des Untertyppolymorphismus.