Wstęp do Programowania potok funkcyjny

Marcin Kubica

2010/2011

Outline

Pojęcie typu

Definition (Typ danych)

- Typ danych to zbiór wartości wraz z zestawem podstawowych operacji na tych wartościach.
- Typy proste i złożone.
- Typy wbudowane.

- Wbudowane typy proste tworzą dziedzinę algorytmiczną. (`bool`, `int`, `float`, `char` i `string`)
- Uwaga na: `+`/`+.`, `*`/`*.` itd.

Pojęcie typu

Definition (Typ danych)

- Typ danych to zbiór wartości wraz z zestawem podstawowych operacji na tych wartościach.
- Typy proste i złożone.
- Typy wbudowane.

- Wbudowane typy proste tworzą dziedzinę algorytmiczną. (`bool`, `int`, `float`, `char` i `string`)
- Uwaga na: `+`/`+.`, `*`/`*.` itd.

Pojęcie typu

Definition (Typ danych)

- Typ danych to zbiór wartości wraz z zestawem podstawowych operacji na tych wartościach.
- Typy proste i złożone.
- Typy wbudowane.

- Wbudowane typy proste tworzą dziedzinę algorytmiczną. (`bool`, `int`, `float`, `char` i `string`)
- Uwaga na: `+`/`+.`, `*`/`*.` itd.

Konstruktory

Definition

Konstruktory to szczególnego rodzaju operacje:

- tworzące wartości określonych typów złożonych,
- różnowartościowe,
- rozłączne przeciwdziedziny,
- razem łącznie są „na” — wszystkie wartości typu złożonego można przedstawić w postaci wyrażenia złożonego z konstruktorów i stałych innych typów,
- są odwracalne — wzorce.

Wzorce

Definition

Wzorce to specjalny rodzaj „wyrażeń”:

- służą do rozbijania wartości typów złożonych,
- zbudowane są z konstruktorów i identyfikatorów,
- *dopasowujemy* do nich wartości,
- jeśli wartość „pasuje”, to identyfikatory z wzorca uzyskują wartości — odpowiadających im składowych dopasowywanej wartości.

Wzorce to podstawowy mechanizm dekompozycji wartości złożonych typów, ale nie tylko.

Wzorce, c.d.

Definition

Składnia wzorców:

$$\langle \texttt{wzorzec} \rangle ::= \langle \texttt{identyfikator} \rangle \mid \_ \mid \langle \texttt{stała} \rangle \mid \ldots$$

- identyfikator — pasuje do każdej wartości, nazwie przyporządkowywana jest dana wartość,
- _ — tak jak identyfikator, tylko nie wprowadza żadnej nowej nazwy,
- stała (np. liczbowa) — pasuje tylko do samej siebie.

Identyfikatory występujące we wzorcu nie mogą się powtarzać.

- Wzorca _ używamy wówczas, gdy interesuje nas tylko, że wartość pasuje do wzorca, a nie chcemy definiować nowych stałych.
- Poznając kolejne typy złożone poznamy inne konstruktory i wzorce.

Wzorce, c.d.

Definition

Składnia wzorców:

$$\langle \texttt{wzorzec} \rangle ::= \quad \langle \texttt{identyfikator} \rangle \mid \_ \mid \langle \texttt{stała} \rangle \mid \ldots$$

- identyfikator — pasuje do każdej wartości,
  nazwie przyporządkowywana jest dana wartość,
- _ — tak jak identyfikator,
  tylko nie wprowadza żadnej nowej nazwy,
- stała (np. liczbowa) — pasuje tylko do samej siebie.

Identyfikatory występujące we wzorcu nie mogą się powtarzać.

- Wzorca _ używamy wówczas, gdy interesuje nas tylko, że wartość pasuje do wzorca, a nie chcemy definiować nowych stałych.
- Poznając kolejne typy złożone poznamy inne konstruktory i wzorce.

Wzorce, c.d.

Definition

Składnia wzorców:

$$\langle \texttt{wzorzec} \rangle ::= \langle \texttt{identyfikator} \rangle \mid \_ \mid \langle \texttt{stała} \rangle \mid \ldots$$

- identyfikator — pasuje do każdej wartości, nazwie przyporządkowywana jest dana wartość,
- _ — tak jak identyfikator, tylko nie wprowadza żadnej nowej nazwy,
- stała (np. liczbowa) — pasuje tylko do samej siebie.

Identyfikatory występujące we wzorcu nie mogą się powtarzać.

- Wzorca _ używamy wówczas, gdy interesuje nas tylko, że wartość pasuje do wzorca, a nie chcemy definiować nowych stałych.
- Poznając kolejne typy złożone poznamy inne konstruktory i wzorce.

Wyrażenia `match-with` i wzorce w definicjach

Definition

Wzorce mogą pojawiać się w:

- wyrażeniach `match-with`,
- definicjach procedur,
- λ-abstrakcji.

$$
\begin{aligned}
\langle \texttt{definicja} \rangle ::= \quad & \underline{\texttt{let}}\ \langle \texttt{wzorzec} \rangle\ \underline{=}\ \langle \texttt{wyrażenie} \rangle \mid \\
& \underline{\texttt{let}}\ [\ \underline{\texttt{rec}}\ ]\ \langle \texttt{identyfikator} \rangle\ \{\ \langle \texttt{wzorzec} \rangle\ \}^*\ \underline{=} \\
& \quad \langle \texttt{wyrażenie} \rangle \\
\langle \texttt{wyrażenie} \rangle ::= \quad & \underline{\texttt{match}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{with}} \\
& \quad \{\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\mid}\ \}^* \\
& \quad\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle \mid \\
& \underline{\texttt{function}}\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle \\
& \qquad \{\ \underline{\mid}\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \}^*
\end{aligned}
$$

Wzorce w definicjach stałych

Wartość wyrażenia po prawej stronie = jest dopasowywana do wzorca po lewej stronie.

Example

```
 3;;
- : int = 3

let _ = 3;;
- : int = 3

let a = 42;;
val a : int = 42
```

Wzorce w definicjach stałych

Wartość wyrażenia po prawej stronie = jest dopasowywana do wzorca po lewej stronie.

Example

```
 3;;
- : int = 3

let _ = 3;;
- : int = 3

let a = 42;;
val a : int = 42
```

Wzorce w definicjach procedur

- Nazwa procedury musi być identyfikatorem.
- Parametry formalne mogą być wzorcami.
- Wartości argumentów wywołania są dopasowywane do parametrów.

Example

```
let f 6 9 = 42;;
val f : int -> int -> int

f 6 9;;
- : int = 42

let p x _ = x + 1;;
val p : int -> 'a -> int = <fun>

p 6 9;;
- : int = 7
```

Wzorce w definicjach procedur

- Nazwa procedury musi być identyfikatorem.
- Parametry formalne mogą być wzorcami.
- Wartości argumentów wywołania są dopasowywane do parametrów.

Example

```
let f 6 9 = 42;;
val f : int -> int -> int

f 6 9;;
- : int = 42

let p x _ = x + 1;;
val p : int -> 'a -> int = <fun>

p 6 9;;
- : int = 7
```

Wyrażenia `match-with`

- Wartość wyrażenia po `match` jest dopasowywana do kolejnych wzorców.
- Pierwszy wzorzec, do którego pasuje, wskazuje wyrażenie, którego wartość jest wartością całego wyrażenia
- Jeżeli żaden wzorzec nie pasuje, to zgłaszany jest błąd (wyjątek).

Example

```
let rec silnia n =
  match n with
    0 -> 1 |
    x -> x * silnia (x-1);;
```

Wyrażenia `match-with`

- Wartość wyrażenia po `match` jest dopasowywana do kolejnych wzorców.
- Pierwszy wzorzec, do którego pasuje, wskazuje wyrażenie, którego wartość jest wartością całego wyrażenia
- Jeżeli żaden wzorzec nie pasuje, to zgłaszany jest błąd (wyjątek).

Example

```
let rec silnia n =
  match n with
    0 -> 1 |
    x -> x * silnia (x-1);;
```

Wzorce w λ-abstrakcjach

Dopasowywanie wzorców w λ-abstrakcjach działa podobnie do wyrażeń `match-with`.

Example

```
let rec silnia =
  function
    0 -> 1 |
    x -> x * silnia (x - 1);;
```

Wzorce w λ-abstrakcjach

Wyrażenie `match-with` jest tylko lukrem syntaktycznym rozwijanym do zastosowania λ-abstrakcji.

Example

```
let rec silnia n =
  (function
    0 -> 1 |
    x -> x * silnia (x-1)
  ) n;;
```

Produkty kartezjańskie

Definition

- $t_1 * \cdots * t_n$ = produkt kartezjański $t_1, \ldots, t_n$.
- Konstruktor postaci: $(x_1, \ldots, x_n)$.
- Równość n-tek — jak w matematyce.

$$\langle \texttt{wyrażenie} \rangle ::= \underline{(}\ [\langle \texttt{wyrażenie} \rangle \{\ \underline{,}\ \langle \texttt{wyrażenie} \rangle\ \}^*]\ \underline{)}$$
$$\langle \texttt{wzorzec} \rangle ::= \underline{(}\ [\langle \texttt{wzorzec} \rangle \{\ \underline{,}\ \langle \texttt{wzorzec} \rangle\ \}^*]\ \underline{)}$$

Produkty kartezjańskie

Example

```
(1, 2, 3);;
- : int * int * int = (1, 2, 3)

(42, (6.9, ”ala”));;
- : int * (float * string) = (42, (6.9, "Ala"))

let (x, s) = (4.5, "Ala");;
val x : float = 4.5
val s : string = "Ala"
```

Produkty kartezjańskie

Example

```
let fib n =
  let rec fibpom n =
    if n = 0 then
      (0, 1)
    else
      let (a, b) = fibpom (n - 1)
      in (b, a + b)
  in
    let (a, b) = fibpom n
    in a;;
val fib : int -> int = <fun>
```

Produkty kartezjańskie

Example

```
let para x y = (x, y);;
val para : 'a -> 'b -> 'a * 'b = <fun>

let rzut_x (x, _) = x;;
val rzut_x : 'a * 'b -> 'a = <fun>

let rzut_y (_, y) = y;;
val rzut_y : 'a * 'b -> 'b = <fun>
```

Produkty kartezjańskie

- Produkt kartezjański nie jest łączny.

```
(2, 3, 4);;
- : int * int * int = (2, 3, 4)

(2, (3, 4));;
- : int * (int * int) = (2, (3, 4))

((2, 3), 4);;
- : (int * int) * int = ((2, 3), 4)
```

- Jednoelementowe n-ki wartości typu t to, po prostu, wartości typu t, $(x) = x$.
- Typ `unit` — odpowiednik typu `void`, „jedynka” produktu kartezjańskiego. Jedyną wartością tego typu jest `()`.

Listy

Definition

- t `list` — ciągi elementów typu t
  (skończone lub nieskończone, ale od pewnego miejsca okresowe).
- *Głowa* — pierwszy element listy.
  Ogon — lista złożoną z wszystkich pozostałych elementów.
  Tak jak wąż, każda niepusta lista składa się z głowy i ogona.
- `[]` — konstruktor listy pustej.
  `h::t` — konstruktor listy, której głową jest `h`, a ogonem `t`.
- $[x_1; x_2; \ldots; x_n] = x_1 :: (x_2 :: \cdots :: (x_n :: [])\ldots)$
- $[x_1; \ldots; x_n]@[y_1; \ldots; y_m] = [x_1; \ldots; x_n, y_1; \ldots; y_m]$
  (czas proporcjonalny do n)
- Dwie listy są równe, jeżeli są równej długości i odpowiadające sobie elementy są równe.

Listy

Definition

- t `list` — ciągi elementów typu t
  (skończone lub nieskończone, ale od pewnego miejsca okresowe).
- *Głowa* — pierwszy element listy.
  Ogon — lista złożoną z wszystkich pozostałych elementów.
  Tak jak wąż, każda niepusta lista składa się z głowy i ogona.
- `[]` — konstruktor listy pustej.
  `h::t` — konstruktor listy, której głową jest `h`, a ogonem `t`.
- $[x_1; x_2; \ldots; x_n] = x_1 :: (x_2 :: \cdots :: (x_n :: []) \ldots)$
- $[x_1; \ldots; x_n]@[y_1; \ldots; y_m] = [x_1; \ldots; x_n, y_1; \ldots; y_m]$
  (czas proporcjonalny do n)
- Dwie listy są równe, jeżeli są równej długości i odpowiadające sobie elementy są równe.

Listy

Definition

- t `list` — ciągi elementów typu t
  (skończone lub nieskończone, ale od pewnego miejsca okresowe).
- *Głowa* — pierwszy element listy.
  Ogon — lista złożoną z wszystkich pozostałych elementów.
  Tak jak wąż, każda niepusta lista składa się z głowy i ogona.
- `[]` — konstruktor listy pustej.
  `h::t` — konstruktor listy, której głową jest `h`, a ogonem `t`.
- $[x_1; x_2; \ldots; x_n] = x_1 :: (x_2 :: \cdots :: (x_n :: [])\ldots)$
- $[x_1; \ldots; x_n]@[y_1; \ldots; y_m] = [x_1; \ldots; x_n, y_1; \ldots; y_m]$
  (czas proporcjonalny do n)
- Dwie listy są równe, jeżeli są równej długości i odpowiadające sobie elementy są równe.

Listy

Definition

- t `list` — ciągi elementów typu t
  (skończone lub nieskończone, ale od pewnego miejsca okresowe).
- *Głowa* — pierwszy element listy.
  Ogon — lista złożoną z wszystkich pozostałych elementów.
  Tak jak wąż, każda niepusta lista składa się z głowy i ogona.
- `[]` — konstruktor listy pustej.
  `h::t` — konstruktor listy, której głową jest `h`, a ogonem `t`.
- $[x_1; x_2; \ldots; x_n] = x_1 :: (x_2 :: \cdots :: (x_n :: [])\ldots)$
- $[x_1; \ldots; x_n]@[y_1; \ldots; y_m] = [x_1; \ldots; x_n, y_1; \ldots; y_m]$
  (czas proporcjonalny do n)
- Dwie listy są równe, jeżeli są równej długości i odpowiadające sobie elementy są równe.

Listy

Example

```
[];;
- : 'a list = []

1::2::3::[];;
- : int list = [1; 2; 3]

[1; 2; 3; 4];;
- : int list = [1; 2; 3; 4]

[1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

["To"; "ci"; "dopiero"; "lista"];;
- : string list = ["To"; "ci"; "dopiero"; "lista"]
```

Listy

Example

Procedura obliczająca długość listy:

```
let length l =
  let rec pom a l =
    match l with
      [] -> a |
      _::t -> pom (a+1) t
  in
    pom 0 l;;
val length : 'a list -> int = <fun>
```

Listy

Example

Procedura obliczająca sumę elementów listy:

```
let sumuj l =
  let rec pom a l =
    match l with
      [] -> a |
      h::t -> pom (a+h) t
  in
    pom 0 l;;
val sumuj : int list -> int = <fun>
```

Listy

Example

Procedura przekształcająca listę par, na parę list równej długości:

```
let rec listapar2paralist =
  function
    []         -> ([], []) |
    (x, y)::t ->
      let (l1, l2) = listapar2paralist t
      in (x :: l1, y :: l2);;
val listapar2paralist :
        ('a * 'b) list -> 'a list * 'b list = <fun>
```

Listy nieskończone

- Można definiować listy nieskończone.
- Listy są implementowane jako wskaźnikowe listy jednokierunkowe.
- Lista nieskończona ma postać cyklu z „ogonkiem”.

Listy nieskończone

Example

```
let rec jedynki = 1::jedynki;;
val jedynki : int list = [1; 1; 1; 1; 1; 1; ...]

let rec cykl = 1 :: 0 :: -1 :: 0 :: cykl;;
val cykl : int list = [1; 0; -1; 0; 1; 0; -1; 0; ...]

[1;2;3] @ cykl;;
- : int list = [1; 2; 3; 1; 0; -1; 0; 1; 0; -1; ...]

cykl @ [1;2;3];;
Stack overflow during evaluation (looping recursion?).
```

Moduł `List`

Moduł `List` zawiera wiele poręcznych procedur operujących na listach. `open List;;` lub `List. ...`

- `length` — długość listy
  (działa w czasie proporcjonalnym do długości listy),
- `hd` — głowa (niepustej) listy,
- `tl` — ogon (niepustej) listy,
- `rev` — oblicza listę złożoną z tych samych elementów, ale w odwrotnej kolejności (uwaga: działa w czasie proporcjonalnym do długości listy),
- `nth` — zwraca element z podanej pozycji na liście
  (głowa ma numer 0, uwaga: działa w czasie proporcjonalnym do numeru pozycji plus 1),
- `append` — sklejanie list, działa tak jak `@`, ale nie jest zapisywane infiksowo, tylko tak, jak zwykła procedura.

Moduł List

Example

```
open List;;
length ["To"; "ci"; "dopiero"; "lista"];;
- : int = 4

hd ["To"; "ci"; "dopiero"; "lista"];;
- : string = "To"

tl ["To"; "ci"; "dopiero"; "lista"];;
- : string list = ["ci"; "dopiero"; "lista"]

rev ["To"; "ci"; "dopiero"; "lista"];;
 - : string list = ["lista"; "dopiero"; "ci"; "To"]

nth ["To"; "ci"; "dopiero"; "lista"] 2;;
- : string = "dopiero"

append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

Listy

Example

Wyszukanie *n*-tego elementu listy:

```
let rec nth l n =
  if n = 0 then hd l
  else nth (tl l) (n - 1);;
```

Listy

Example

Odwrócenie listy `rev`:

```
let rev l =
  let rec pom l w =
    if l = [] then w
    else pom (tl l) ((hd l) :: w)
  in pom l [];;
```

Listy

Example

Sklejanie list `@`, append:

```
let rec append l1 l2 =
  if l1 = [] then l2
  else (hd l1) :: (append (tl l1) l2);;
```

Definicje typów

- Nadanie nazw typom złożonym.
  Nowej nazwy typu można używać wymiennie z typem, który oznacza.
- Specyfikacje określające jakiego typu powinny być określone wartości.
  Kompilator stara się używać nazw podanych w takich specyfikacjach.
- Typy sparametryzowane.
  Kolekcje elementów, których typ jest dowolny, określony jako parametr.
  Na przykład listy.
- Niektóre typy złożone trzeba zdefiniować przed użyciem.

Składnia definicji typów

Definition

⟨jedn.kompilacji⟩ ::= ⟨deklaracja typu⟩
⟨deklaracja typu⟩ ::= type ⟨identyfikator⟩ = ⟨typ⟩ |
type ⟨parametr typowy⟩ ⟨identyfikator⟩ = ⟨typ⟩ |
type (⟨parametr typowy⟩ { , ⟨parametr typowy⟩ }*)
⟨identyfikator⟩ = ⟨typ⟩
⟨typ⟩ ::= ⟨identyfikator⟩ |
⟨typ⟩ ⟨identyfikator⟩ |
(⟨typ⟩ { , ⟨typ⟩ }*) ⟨identyfikator⟩ |
⟨typ⟩ { * ⟨typ⟩ }* |
(⟨typ⟩) | ...
⟨parametr typowy⟩ ::= '⟨identyfikator⟩

Definicje typów

Example

```
type p = int * float;;

type 'a lista = 'a list;;

type ('a, 'b) para = 'a * 'b;;

type 'a t = ('a, 'a) para * ('a list) list;;
```

Definition (Składnia specyfikacji typów)

⟨wyrażenie⟩ ::= (⟨wyrażenie⟩ : ⟨typ⟩)
⟨wzorzec⟩ ::= (⟨wzorzec⟩ : ⟨typ⟩)

Definicje typów

Example

```
type p = int * float;;

type 'a lista = 'a list;;

type ('a, 'b) para = 'a * 'b;;

type 'a t = ('a, 'a) para * ('a list) list;;
```

Definition (Składnia specyfikacji typów)

$$\langle \texttt{wyrażenie} \rangle ::= \underline{\texttt{(}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{:}}\ \langle \texttt{typ} \rangle\ \underline{\texttt{)}}$$
$$\langle \texttt{wzorzec} \rangle ::= \underline{\texttt{(}}\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{:}}\ \langle \texttt{typ} \rangle\ \underline{\texttt{)}}$$

Specyfikacje typów

Example

```
type point = float * float;;
type vector = point;;

let (p:point) = (2.0, 3.0);;
val p : point = (2., 3.)

let shift ((x, y): point) ((xo, yo): vector) =
  ((x +. xo, y +. yo) : point);;
val shift : point -> vector -> point = <fun>

shift p p;;
- : point = (4., 6.)
```

Rekordy

- Rekordy są podobne do produktów kartezjańskich.
- Rekord to n-ka wartości określonych typów.
- Współrzędne (pola rekordu) są identyfikowane po nazwie, a nie wg. pozycji w n-ce.
- Rekordy występują w wielu językach programowania. W C i C++ są nazywane strukturami.

Rekordy

Definition

Typy rekordowe wprowadza się deklarując typ postaci:

$$\langle \texttt{typ} \rangle ::= \quad \underline{\texttt{\{}} \; \{ \langle \texttt{identyfikator} \rangle \; \underline{\texttt{:}} \; \langle \texttt{typ} \rangle \; \underline{\texttt{;}} \; \}^* \; \langle \texttt{identyfikator} \rangle \; \underline{\texttt{:}} \; \langle \texttt{typ} \rangle \, [\; \underline{\texttt{;}} \;] \; \underline{\texttt{\}}}$$

Identyfikatory to nazwy pól rekordów.

Uwaga

Ze względu na przysłanianie nazw, nie należy używać tej samej nazwy pola w różnych typach rekordów.
W przeciwnym przypadku, będziemy mogli korzystać tylko z typu, który został zdefiniowany później.

Rekordy

Definition

Typy rekordowe wprowadza się deklarując typ postaci:

$$\langle \texttt{typ} \rangle ::= \quad \underline{\texttt{\{}} \; \{ \langle \texttt{identyfikator} \rangle \; \underline{\texttt{:}} \; \langle \texttt{typ} \rangle \; \underline{\texttt{;}} \; \}^* \; \langle \texttt{identyfikator} \rangle \; \underline{\texttt{:}} \; \langle \texttt{typ} \rangle \; [\; \underline{\texttt{;}} \;] \; \underline{\texttt{\}}}$$

Identyfikatory to nazwy pól rekordów.

Uwaga

Ze względu na przysłanianie nazw, nie należy używać tej samej nazwy pola w różnych typach rekordów.
W przeciwnym przypadku, będziemy mogli korzystać tylko z typu, który został zdefiniowany później.

Rekordy

Definition

Konstruktor wartości rekordowych ma postać:

$$
\begin{aligned}
\langle \texttt{wyrażenie} \rangle ::= \quad & \underline{\{} \, \{ \langle \texttt{identyfikator} \rangle \, \underline{=} \, \langle \texttt{wyrażenie} \rangle \, \underline{;} \, \}^* \\
& \quad \langle \texttt{identyfikator} \rangle \, \underline{=} \, \langle \texttt{wyrażenie} \rangle \, [\, \underline{;} \,] \, \underline{\}} \\
\langle \texttt{wzorzec} \rangle ::= \quad & \underline{\{} \, \{ \langle \texttt{identyfikator} \rangle \, \underline{=} \, \langle \texttt{wzorzec} \rangle \, \underline{;} \, \}^* \\
& \quad \langle \texttt{identyfikator} \rangle \, \underline{=} \, \langle \texttt{wzorzec} \rangle \, [\, \underline{;} \,] \, \underline{\}}
\end{aligned}
$$

Tworząc rekordy musimy podać wartości wszystkich pól (w dowolnej kolejności).
We wzorcu, możemy podać tylko interesujące nas pola.
Dwa rekordy są równe, jeśli odpowiadające sobie pola są równe.

Rekordy

Definition

Do pojedynczych pól rekordów możemy się odwoływać podając nazwę pola po kropce.

$$\langle \texttt{wyrażenie} \rangle ::= \langle \texttt{wyrażenie} \rangle \ \texttt{.} \ \langle \texttt{identyfikator} \rangle$$

Rekordy

Example

```
type ułamek = { licznik : int ; mianownik : int };;

let q = { licznik = 3; mianownik = 4 };;
val q : ulamek = {licznik = 3; mianownik = 4}

let { licznik = a ; mianownik = b } = q;;
val a : int = 3
val b : int = 4

let { licznik = c } = q;;
val c : int = 3

q.licznik;;
- : int = 3
```

Typy wariantowe

Uogólnienie obejmujące typy wyliczeniowe, unie i drzewa.

Definition

Typy wariantowe deklarujemy w następujący sposób:

$$\begin{array}{lll} \langle \texttt{typ} \rangle & ::= & \langle \texttt{wariant} \rangle \{ \ \underline{|} \ \langle \texttt{wariant} \rangle \}^* \\ \langle \texttt{wariant} \rangle & ::= & \langle \texttt{Identyfikator} \rangle [\ \underline{\texttt{of}} \ \langle \texttt{typ} \rangle] \end{array}$$

$\langle$`Identyfikator`$\rangle$ oznacza identyfikatory rozpoczynające się wielką literą.
$\langle$`identyfikator`$\rangle$ oznacza identyfikatory rozpoczynające się małą literą.

Każdy wariant wprowadza konstruktor o podanej nazwie.
Konstruktor taki może mieć co najwyżej jeden argument,
ale zawsze może to być argument odpowiedniego typu produktowego.

Typy wariantowe

Uogólnienie obejmujące typy wyliczeniowe, unie i drzewa.

Definition

Typy wariantowe deklarujemy w następujący sposób:

$$\begin{array}{lcl} \langle \texttt{typ} \rangle & ::= & \langle \texttt{wariant} \rangle \{ \ \underline{|} \ \langle \texttt{wariant} \rangle \}^* \\ \langle \texttt{wariant} \rangle & ::= & \langle \texttt{Identyfikator} \rangle [\ \underline{\texttt{of}} \ \langle \texttt{typ} \rangle] \end{array}$$

$\langle$`Identyfikator`$\rangle$ oznacza identyfikatory rozpoczynające się wielką literą.
$\langle$`identyfikator`$\rangle$ oznacza identyfikatory rozpoczynające się małą literą.

Każdy wariant wprowadza konstruktor o podanej nazwie.
Konstruktor taki może mieć co najwyżej jeden argument,
ale zawsze może to być argument odpowiedniego typu produktowego.

Typy wariantowe

Definition

Konstruktorów typów wariantowych używamy w następujący sposób:

$$\begin{aligned} \langle \texttt{wyrażenie} \rangle &::= \langle \texttt{Identyfikator} \rangle \, [\, \langle \texttt{wyrażenie} \rangle \,] \\ \langle \texttt{wzorzec} \rangle &::= \langle \texttt{Identyfikator} \rangle \, [\, \langle \texttt{wzorzec} \rangle \,] \end{aligned}$$

Wartości typu wariantowego są sobie równe, jeżeli:

- są wynikiem tego samego konstruktora, oraz
- oba argumenty konstruktora są sobie równe.

Typy wariantowe

Example

```
type znak = Dodatni | Ujemny | Zero;;
type znak = Dodatni | Ujemny | Zero

let nieujemny x =
  match x with
    Ujemny -> false |
    _      -> true;;
val nieujemny : znak -> bool = <fun>

let ujemny x =
  x = Ujemny;;
val ujemny : znak -> bool = <fun>
```

Typy wariantowe

Example

```
type zespolone =
  Prostokatny of float * float |
  Biegunowy of float * float ;;
type zespolone = Prostokatny of float * float | Biegunowy
of float * float

let modul =
  function
    Prostokatny (x, y) -> sqrt (square x +. square y) |
    Biegunowy (r, _) -> r;;
val modul : zespolone -> float = <fun>
```

Drzewa

- Deklaracje typów wariantowych mogą być rekurencyjne.
- W ten sposób możemy konstruować drzewa.

Example

```
type drzewo =
  Puste |
  Wezel of int * drzewo * drzewo;;
```

- Można tworzyć „zapętlone" drzewa — podobnie jak w przypadku list.

Example

```
let rec t = Wezel (42, t, Puste);;
val t : drzewo = Wezel (42, Wezel (...), Puste)
```

Drzewa

- Deklaracje typów wariantowych mogą być rekurencyjne.
- W ten sposób możemy konstruować drzewa.

Example

```
type drzewo =
  Puste |
  Wezel of int * drzewo * drzewo;;
```

- Można tworzyć „zapętlone" drzewa — podobnie jak w przypadku list.

Example

```
let rec t = Wezel (42, t, Puste);;
val t : drzewo = Wezel (42, Wezel (...), Puste)
```

Sparametryzowane typy wariantowe

- Definicje typów wariantowych mogą być sparametryzowane.
- Parametr może określać typ elementów struktury danych.

Example

```
type 'a lista = Pusta | Pelna of 'a * 'a lista;;
type 'a lista = Pusta | Pelna of 'a * 'a lista

Pelna (4, Pusta);;
- : int lista = Pelna (4, Pusta)

Pelna ("ala", Pelna("ula", Pusta));;
- : string lista = Pelna ("ala", Pelna ("ula", Pusta))

Pelna (4, Pelna ("ula", Pusta));;
error ...
```

Aliasy we wzorcach

Definition

Dopasowywana wartość jest przypisywana identyfikatorowi po prawej stronie as oraz jest dopasowywana do wzorca po lewej stronie.

$$\langle \texttt{wzorzec} \rangle ::= \langle \texttt{wzorzec} \rangle \ \underline{\texttt{as}} \ \langle \texttt{identyfikator} \rangle$$

Przydatne, gdy chcemy uchwycić zarówno całość, jak i część struktury danych.

Aliasy we wzorcach

Example

```
let (x, y) as para = (2, 3);;
val x : int = 2
val y : int = 3
val para : int * int = (2, 3)

let (h::t) as lista = [1; 2; 3; 4];;
val h : int = 1
val t : int list = [2; 3; 4]
val lista : int list = [1; 2; 3; 4]
```

Wyjątki

- Co zrobić jeśli wynik procedury jest nieokreślony?
  Jak zgłosić błąd?
- Należy użyć *wyjątków*.
- Wyjątek, to wartość specjalnego typu wariantowego `exn`.
  Wartości tego typu niosą informacje o wyjątkowych sytuacjach, przyczynach uniemożliwiających wykonanie obliczeń.
- Typ `exn` możemy rozszerzać o nowe warianty:

Definition

⟨jednostka kompilacji⟩ ::= ⟨deklaracja wyjątku⟩ | ...
⟨deklaracja wyjątku⟩ ::= exception ⟨wariant⟩

Zgłaszanie wyjątków

- Wyjątki można *zgłaszać* (*podnosić*) i *przechwytywać*.
- Zgłoszenie wyjątku, to wyrażenie, którego obliczenie „nie udaje się", a wyjątek niesie informację o przyczynie porażki.
- Jeśli obliczenie podwyrażenia powoduje zgłoszenie wyjątku, to tak samo dzieje się z całym wyrażeniem.
  Zgłoszony wyjątek, to rodzaj propagującego się „błędu w obliczeniach".

Definition

Podniesienie wyjątku ma następującą postać:

$$\langle \texttt{wyrażenie} \rangle ::= \quad \underline{\texttt{raise}}\ \langle \texttt{wyrażenie} \rangle$$

Argument operacji `raise` musi być typu `exn`.

Przechwytywanie wyjątków

- Przechwycenie wyjątku to konstrukcja odwrotna do jego zgłoszenia.
- Możemy spróbować obliczyć wyrażenie, licząc się z możliwością zgłoszenia wyjątku.
- Jeśli wyjątek zostanie zgłoszony, to możemy go przechwycić i podać „zastępczą" wartość wyrażenia.

Definition

$$\begin{aligned}\langle \texttt{wyrażenie} \rangle ::= \quad & \underline{\texttt{try}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{with}} \\ & \{\, \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{|}}\ \}^* \\ & \quad \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\end{aligned}$$

Wyjątki

Example

```
exception Dzielenie_przez_zero of float;;
exception Dzielenie_przez_zero of float

let dziel x y =
  if y = 0.0 then
    raise (Dzielenie_przez_zero x)
  else x /. y;;
val dziel : float -> float -> float = <fun>
```

Wyjątki

Example

```
let odwrotnosc x =
  try
    dziel 1.0 x
  with
    Dzielenie_przez_zero _ -> 0.0;;
val odwrotnosc : float -> float = <fun>

odwrotnosc 42.;;
- : float = 0.023809523809523808

odwrotnosc 0.0;;
- : float = 0.
```

Wyjątki

- Standardowo zdefiniowane są:
  - wyjątek `Failure of string`, oraz
  - oraz procedura:

```
let failwith s =
  raise (Failure s);;
```

- Należy odróżniać wartości typu `exn` od zgłoszenia wyjątku:

```
Dzielenie_przez_zero 4.5;;
- : exn = Dzielenie_przez_zero 4.5
raise (Dzielenie_przez_zero 4.5);;
Exception: Dzielenie_przez_zero 4.5.
```