TKOM - Dokumentacja

Jeremi Sobierski 310901

Opis funkcjonalności

Typy

W języku korzystać można zarówno z typów wbudowanych jak i zdefiniowanych przez użytkownika. Definicje typów mogą być generyczne. Każdy nowo zdefiniowany typ posiada prototyp, który definiuje dane w nim zawarte. Prototyp może być dowolnym innym typem).

Proste typy wbudowane:

1. `int`
2. `bool`
3. `char`
4. `float`
5. `String`

Złożone typy wbudowane:

1. krotka - `(...)`
2. wektor - `[...]`
3. strukutra - `{...}`

```
type Angle float

type Car {
    model: std::String,
    doors: int,
    fuel: float,
}
```

definicja własnego typu

Implementacje

Użytkownik może definiować metody na typach własnych:

```
def Car {
    fun getFuel(): float -> this.fuel
    fun hasFourDoors(): bool -> this.doors == 4
}
```

Cechy

Użytkownik może definiować własne cechy oraz je implementować. Może również korzystać z implementacji do przeciążania pewnych operatorów.

```
trait Vehicle {
    fun drive()
    fun getFuel(): float
}

def Car as Vehicle {
    fun drive() { ... }
    fun getFuel(): float -> ...
}
```

definicja cechy oraz jej implementacja

Funkcje

Język pozwala na definicje nowych, potencjalnie generycznych funkcji. Argumenty do funkcji przekazywane są przez kopię.

```
fun testVehicle<V is Vehicle>(vehicle: V) {
    vehicle.drive();
    print(
        "fuel after driving: " + vehicle.getFuel() as std::String
    );
}
```

Wywołanie funkcji lub metody odbywa się z użyciem nawiasów. Jeżeli metoda o podanej nazwie występuje w jendej implementowanej cesze, to należy jawnie wskazać jedną z nich.

```
register(car);
car.setFuel(42.0);
car.crashInto:<Truck>();

// assuming dive is also in another implementation
car.<Vehicle>drive();
```

Argumenty generyczne można pominąć, jeżeli jest możliwość ich domniemania na podstawie argumentów wywołania:

```
fun first<T>(items: [T]): T -> items[0]

// first:<int> inferred
let a = first([1, 2, 3]);
```

Ograniczone implemetacje

Implementacje także mogą zawierać ograniczenia:

```
trait Show {
    fun show()
}

type Point {
    x: int,
    y: int,
}

def<T is Show> [T] as Show {
    fun show() {
        var items = this;
        while (let [a, b, ..rest] = items) {
            a.show();
            ", ".print();
            items = [b, ..rest];
        };
        if (items.length() > 0) {
            items[0].show();
        };
    }
}

def Point as Show {
    fun show() {
        (
            "(" + this.x as String
                + ", "
                + this.y as String
                + ")"
        ).print();
    }
}

fun main(): int {
    let point = Point { x: 0, y: 0 };
    [point, point, point].show();
    0
}
```

Implementacje dopuszczają także specjalizacje:

```
def<T is Show> [T] as Show {
    fun show() { ... }
}

def Point as Show {
    fun show() {
        (this.x as String + ", " + this.y as String).print();
    }
}

// only print the parenthases when showing a vector of points
def [Point] as Show {
    fun show() {
        var items = this;
        while (let [a, b, ..rest] = items) {
            "(".print();
            a.show();
            "), ".print();
            items = [b, ..rest];
        };
        if (items.length() > 0) {
            "(".print();
            items[0].show();
            ")".print();
        };
    }
}
```

Dobór specjalizacji musi być rozstrzygalny, nawet jeśli typ który powodowałby niejednoznaczność jeszcze nie istnieje:

```
def<T> [T] as Show { ... } // ok
def<T is Show> [T] as Show { ... } // ok
def<T is std::Add<T>> [T] as Show { ... } // error
def<T is std::Add<T> & Show> [T] as Show { ... } // ok
```

```
invalid specialization: does not specialize an existing implementation
  |
67| def <T is std::Add<T>> [T] as Show {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  @ test.ice:67:1

note: this implementation is equally specific
  |
47| def<T is Show> [T] as Show {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^
  @ test.ice:47:1
```

Wyrażenia

Poza oczekiwanymi wyrażeniami (literały, wywołania itd.) dostępne są także następujące konstrukcje:

`if` - na podstawie warunku logicznego ewaluuje wartość jednego z dwóch wyrażeń. Jeśli oba mają ten sam typ, przyjmuje wartość ewaluowanego wyrażenia.

`while` - wielokrotnie ewaluuje wyrażenie na podstawie warunku logicznego lub dopasowania. Dopasowane zmienne dostępne są w ciele pętli.

`match` - dopasowuje wartość wyrażenia do jednego z wzorców, a następnie ewaluuje i przyjmuje wartość przypożądkowanego dopasowanemu wzorcowi wyrażenia. Jeśli wzorzec zawiera zmienną o niezwiązanej nazwie (lub w przypadku dopasowania do struktury o nazwie takiej jak jedno z pól), to przypisywania jest do niej dopasowywana wartość. Dodatkowo jeśli wzorzec jest wyrażeniem złożonym, to jego wartość musi być prawdziwa. Jeśli wzorzec nie zawiera niezwiązanych zmiennych to dopasowywana wartość musi być równa wzorcowi. Przy dopasowaniu wektora można również jednokortnie skorzystać z ".." aby dopasować wiele elementów.

```
let result = if (condition) {
    print("was true");
    0
} else 1;

while (index < 10) {
    print("loop");
    index = index + 1;
};

let nums = [0, 0, 0, 1, 2, 3];

let (a, b) = (1, 2);

var vector = nums;
while (let [0, ..rest] = vector) { vector = rest; };
// vector == [1, 2, 3]

let message = match car {
    { fuel == 0.0, model }       => model + " has no more fuel!",
    { 0.0 < fuel && fuel < 0.2 } => "low on fuel",
    { fuel }                     => {
        fuel as std::String + " fuel left"
    },
};
```

Zmienne

Język pozwala definiować zmienne mutowalne za pomocą słowa kluczowego `var` oraz niemutowalne za pomocą słowa kluczowego `let`. Typ zmiennej może być pominięty, zostanie on wtedy domniemany.

```
let num    = 3;
var mutNum = 4;

num    = 2; // error: let bindings cannot be reassigned
mutNum = 2; // OK
```

Zmienne nie współdzielą zasobów:

```
var nums  = [1, 2, 3];
var nums2 = nums;
nums[0] = 0;
// nums2[0] == 1;
```

Gramatyka

Poniżej przedstawiona jest gramatyka języka

```
program             = { declaration };

declaration         = functionDefinition
                    | typeDefinition
                    | traitDefinition
                    | traitImplementation;
                    | implementation;

functionDefinition  = signature, ( "->", expression | block );

typeDefinition      = [ visibility ], "type", identifier, [ typeParameters ],
typeName;
traitDefinition     = [ visibility ], "trait", identifier, [ typeParameters ],
traitBody;
implementation      = "def", [ typeParameters ], typeName , implementations;
traitImplementation = "def", [ typeParameters ], typeName, "as", traitName
                    , implementations;


nameAndType         = identifier, ":", typeName;

typeParameters      = "<", typeParameter, { ",", typeParameter }, [","], ">";
typeArguments       = "<", typeName, { ",", typeName }, [","], ">";
typeParameter       = identifier, [ "is", traitName, { "+", traitName } ];

typeName            = identifier, [ typeArguments ]
                    | "(", [  typeName, { ",",  typeName }, [","] ], ")"
                    | "{", [ nameAndType, { ",", nameAndType }, [","] ], "}"
                    | "[", typeName ,"]";

traitName           = identifier, [ typeArguments ];
implementations     = "{", { functionDefinition }, "}"
                    | functionDefinition;


visibility          = "public" | "internal"
signature           = [ visibility ], "fun", identifier, [ typeParameters ], "("
                    , [ nameAndType, { ",", nameAndType }, [","] ]
                    , ")", [ ":", typeName ];

traitBody           = "{", { signature }, "}"
                    | signature;

tupleExpression     = expression, { ",", expression };
expression          = orExpression, [ "=", expression ];
orExpression        = andExpression, { "||", andExpression };
andExpression       = equalityExpression , { "&&", equalityExpression };

equalityExpression  = relationalExpression
                    , { equalityOperator, relationalExpression };
```

```
relationalExpression     = additiveExpression
                         , { relationalOperator, additiveExpression };

additiveExpression       = multiplicativeExpression
                         , { additiveOperator, multiplicativeExpression };

multiplicativeExpression = castExpression
                         , { multiplicativeOperator, castExpression };

castExpression           = prefixExpression, { "as", typeName };
prefixExpression         = { prefixOperator }, postfixExpression;
postfixExpression        = primaryExpression, { postfixOperator };
primaryExpression        = stringLiteral
                         | numericLiteral
                         | structLiteral
                         | identifier, [ structLiteral ]
                         | vectorLiteral
                         | "(", expression, ")"
                         | "(", [ expression, ",", [ tupleExpression, [","] ] ],")"
                         | "match", scrutinee, "{" matchCase, { matchCase } "}"
                         | "if", "(", condition, ")", expression, "else"
                         | "while", "(", condition, ")";
                         | block;

vectorLiteral           = "[", ":", typeName", "]"
                        | "[", expression, { ",", expression }, [","] "]"

structLiteral           =  "{", [ structProperty, { ",", structProperty },
[","], ], "}"
structProperty          = identifier, [ ":", expression ];

block                   = "{", blockItem, ";", [ blockItem, { ";", blockItem } ],
[ ";" ] "}"

scrutinee               = identifier
                        | vectorLiteral
                        | "(", expression, ")"
                        | "(", expression, ",", [ tupleExpression, [","] ], ")";
                        // + dowolne operatory, tak jak w expression

matchCase               = tuplePattern, "=>", matchArm, ";";

matchArm                = tupleExpression
                        | exitFlowOperator, [ tupleExpression ];

prefixOperator          = "!" | "-";
postfixOperator         = ".", identifier, [ call ]
                        | ".", numericLiteral,
                        | ".", "match", "{", matchCase, { matchCase }, "}"
                        | ".", "<", traitName, ">", identifier, call,
                        | "[", tupleExpression, "]"
                        | call;

call                    = [ callTypeArguments ]
                        , "(" [ expression, { ",", expression }, [","] ] ")";

callTypeArguments       = ":<", typeName, { ",", typeName }, [","], ">";
```

```
blockItem   = "let" , destructureTuple, [ ":", typeName ], "=", tupleExpression
            | "var" , destructureTuple, [ ":", typeName ], "=", tupleExpression
            | exitFlowOperator, [ tupleExpression ]
            | expression;

equalityOperator        = "==" | "!=";
relationalOperator      = "<" | ">" | "<=" | ">=";
multiplicativeOperator  = "/" | "*" | "%";
additiveOperator        = "+" | "-";
exitFlowOperator        = "return" | "break" | "continue";

condition               = "let", tuplePattern, [":",typeName ],"=",tupleExpression;
                        | expression;

tuplePattern            = pattern, { ",", pattern };
patternBody             = destructure | guardExpression;
pattern                 = destructure, [ "if", orExpression ]
                        | guardExpression;

guardExpression         = identifier | stringLiteral | numericLiteral
                        // + dowolne operatory, tak jak w expression

destructureTuple        = destructure, { ",", destructure };

destructure     = "(" destructureTuple, [","] ")"
                | "{", destructureProperty, { ",", destructureProperty }, [","] "}"
                | "[", destructureElement, { ",", destructureElement }, [","] "]";

destructureProperty     = patternBody, [ ":", destructure ];
destructureElement      = destructure
                        | "..", [ identifier ];
```

Tokeny:

```
identifier      matches "[a-zA-Z][a-zA-Z0-9]*";
numericLiteral  matches "\d+(\.\d+)?";
stringLiteral   matches "\".*?[^\\]\"";
keywords        = "fun" | "type" | "def" | "trait" | "var" | "let" | "int"
                | "float" | "as" | "is" | "for" | "if" | "while" | "else"
                | "return" | "break" | "match";

punctuation     = ".." | "." | ":" | ":<" | "," | "<" | ">" | "<=" | ">=" | "=="
                | "!=" | "/" | "*" | "%" | "+" | "-" | "(" | ")" | "{" | "}" | "["
                | "]" | "=" | "!" | "&&" | "||";
```

Komunikaty Błędów

Przykłady komunikatów błędów napotkanych w trakcie kompilacji lub wykonania programu:

```
type error: expected value of type "String", has type "int" instead
  |
42| let text: String = 456;
  |                    ^^^
  @ main.ice:42:21

syntax error: expected ";", got "fun" instead
  |
42| let text: String = 456 fun
  |                        ^^^
  @ main.ice:42:25

index out of bounds error: accessing index 5 on a vector of size 3
  |
36| let element = vector[5];
  |                     ^^^
  @ main.ice:36:22

arithmetic error: division by zero
  |
70| fraction = 3 / 0;
  |              ^^^
  @ main.ice:70:14
```

Sposób uruchomienia

Poniżej przykład komendy skompilowania oraz uruchomienia programu składającego się z dwóch plików źródłowych:

```
icy main.ice lib.ice
clang++ iout.o runtime.cpp -o outputFile
./outputFile
```

Zwięzły opis realizacji

Główne komponenty

- lekser
- parser
- analiator sematyczny
- generator formy pośredniej LLVM

Lekser przekazuje strumień tokenów do parsera, który buduje na jego podstawie AST. Przechodzi przez nie następnie analizator sematyczny, sprawdzając poprawność typów, ograniczeń typów generycznych oraz odnajduje symbole. Po tym przez drzewo przechodzi generator formy pośredniej LLVM, która następnie kompilowana jest do pliku objektowego.

Zwięzły opis sposobu testowania

Lekser testowany jest bezpośrednio poprzez testy jednostkowe weryfikujące poprawność generowanych sekwencji tokenów. Parser testowany jest z pomocą leksera - w testach jednostkowych porównywana jest tekstowa reprezetnacja drzewa obiektów generowanych przez parser na podstawie fragmentu kodu źródłowego. Dodatkowo testownana jest też część pomocnicznych funkcji parsera. Kompilator testowany jest z pomocą leksera oraz parsera - fragmenty kodu są kompilowane a następnie wykonywane. Funkcje zewnętrzne wykorzystywane są do zweryfikowania wartości w trakcie wykonywania fragmentu. Na koniec sprawdzany jest rezultat kompilacji oraz wykonania.