Nemerle

Tadeusz Sznuk

2006.XII.18

Plan prezentacji

(i) Wprowadzenie

(ii) Programowanie w Nemerle

(iii) Metaprogramowanie

Część I: Wprowadzenie

Cechy języka

Nemerle jest wysokopoziomowym językiem działającym na platformie . NET, stworzonym na Uniwersytecie Wrocławskim. Jego główne cechy to

- Możliwość połączenia stylu obiektowego i funkcyjnego
- Dobra integracja z `.NET`.
- Statyczny system typów.
- Rozwinięty mechanizm metaprogramowania.
- Prosta składnia podobna do C#.

Czym NIE jest Nemerle

Nemerle nie jest

- Kolejnym językiem z rodziny pytonowatych.
- .NET´ową wersją innego języka (jak SML.NET czy F#).
- Makra w Nemerle nie mają wiele wspólnego z prymitywnym mechanizmem znanym z języka C.

Część II: Programowanie w Nemerle

3. Składnia
   - Definiowanie klas
   - Funkcje lokalne
   - Krotki, listy, warianty
   - Dopasowanie wzorca
   - ...
4. Typy
   - System typów
   - Algorytm wnioskowania typu
5. Narzędzia
   - Edytory
   - Cs2n
   - Inne
6. Inne kwestie
   - Wywołania ogonowe

Definicja klasy

Kod przykładowej klasy

```
namespace My.Namespace {
    public class MyClass {

        x : int;

        public senseOfLife () : int
        {
            x
        }

        public this ()
        {
            x = 42;
        }

        public this (x : int)
        {
            this.x = x;
        }
    }
}
```

Definicja klasy

- Typy pól, parametrów i wyników fukcji piszemy po dwukropku.
- Konstruktory deklarujemy jako metody o nazwie `this`. Jeśli żadnego nie podamy, powstanie domyślny.
- Domyślnie wartość pola może być ustawiona jedynie w kodzie inicjalizującym

```
public x : int = 42;
```

  lub w konstruktorze, jak w przykładowej klasie. Jeśli chcemy mieć możliwość zmieniania wartości pola musimy dodać modyfikator `mutable`.
- Pola i metody domyślnie są prywatne.
- Tworząc obiekt podajemy tylko nazwę klasy i parametry konstruktora, bez słowa `new`.

Definicja klasy

- Mamy dostęp do modyfikatorów znanych z C# (np. `protected`, `private`, `internal`, `volatile`).
- Możemy również używać `.NET´owych` atrybutów

Atrybuty

```
[Serializable]
public class MyClass {
...
```

- Identycznej składni mogą uzywać makra - nie wszystko atrybut, co się świeci.

Nemerle

Tadeusz Sznuk

Konstruktory statyczne

Jeśli zadeklarujemy konstruktor jako statyczny, zostanie on wywołany przed pierwszym odwołaniem do klasy, która go zawiera.

Konstruktor statyczny

```
public class Program {

    static this ()
    {
        System.Console.WriteLine ("ZPO");
    }

    public static Run () : void
    {
        System.Console.WriteLine ("MPO");
    }
}

System.Console.WriteLine ("PO");
Program.Run ();

/* WYNIK:
   PO
   ZPO
   MPO
*/
```

Właściwości

Właściwości możemy definiować tak jak w C#.

```
public class C {

    mutable _x : int = 42;

    public X : int {
        get { _x }
        set { _x = value }
    }
}
```

Możemy także użyć makra [Accessor].

```
using Nemerle.Utility;

public class C {

    [Accessor (flags = WantSetter)]
    mutable _x : int = 42;

}
```

Definiowanie klasy w kilku plikach

Przy pomocy słowa kluczowego `partial` możemy rozbić definicję klasy na kilka plików.

partial1.n

```
public partial class C {
    public A () : int
    {
        42
    }
}
```

partial2.n

```
public partial class C {
    public B () : int
    {
        840826
    }
}
```

Dziedziczenie

- Stosujemy modyfikatory z C# (`virtual`, `override`, `abstract`, ...).
- Domyślnie metody *nie* są wirtualne.
- Słowo kluczowe base ma znaczenie takie jak w C#.
- Nie ma wielodziedziczenia.
- Możemy dziedziczyć po dowolnej `.NET`´owej klasie.

Przykład dziedziczenia

```
public class Superclass {
    protected this (_ : int) { }
}

interface Interface {
    Method (_ : int) : void;
}

public class Subclass : Superclass, Interface {
    public this ()
    {
        base (42);
    }

    public Method (_ : int) : void { }
}
```

Moduły

Moduł zachowuje się jak klasa bez konstruktorów, której wszystkie pola i metody są statyczne.

Przykład modułu

```
public module App {

    x : int = 42;

    public Main () : void
    {
        System.Console.WriteLine ($ "x = $x");
    }

}
```

Nemerle

Tadeusz Sznuk

Metody rozszerzające

Możemy je definiować w sposób podobny jak w C# (3.0).

Przykład

```
namespace ExtensionExample {
  class ExtensionClass {
    public static Next (this i : int) : int {
      i + 1
    }
  }
}
using ExtensionExample;
def x = 41;
System.Console.WriteLine (x.Next ());
```

Funkcje lokalne

Nemerle umożliwia tworzenie funkcji lokalnych oraz używanie ich jak wartości.

Przykład funkcji lokalnej

```
using Nemerle.Collections;

public module Mod {
    public Odd (l : list [int]) : list [int] {
        def isOdd (x) {
            x % 2 == 0
        }
        List.Filter (l, isOdd)
    }
}
```

Funkcje anonimowe

Możemy zdefiniować i wykorzystać funkcję bez nazwy.

Funkcja anonimowa

```
using Nemerle.Collections;

def rev (l) {
    List.FoldLeft (l, [], fun (el, acc) { el :: acc })
}
```

Składnia podobna do C# też jest możliwa.

Funkcja anonimowa w stylu C#

```
using Nemerle.Collections;
def rev (l) {
    List.FoldLeft (l, [], (el, acc) => { el :: acc })
}
```

Wywołując funkcję możemy pominąć niektóre argumenty, wstawiając zamiast nich znak _. Wartością takiego wrażenie jest funkcja, której parametrami są pominięte argumenty.

Przykład

```
using Nemerle.Collections;

public module Mod {
    public Even (l : list [int]) : list [int] {
        def divides (x, y) { y % x == 0 }
        List.Filter (l, divides (2, _))
    }
    public Inc (l : list [int], d : int) : list [int]
    {
        List.Map (l, _ + d)
    }
}
```

Domyślne parametry

Możemy dla niektórych parametrów podać domyślne wartości. Należy jednak zauważyć, że podane po = wyrażenie będzie wyliczane przy każdym wywołaniu funkcji.

Parametry domyślne

```
def rev (l, acc = []) {
  match (l) {
    |x :: xs => rev (xs, x :: acc)
    |[] => acc
  }
}
rev ([1,2,3])
```

Nazwane parametry

Jeśli wywołując funkcję podamy nazwy parametrów, możemy zaburzyć ich kolejność. Możemy też pominąć dowolne z nich (o ile mamy wartości domyślne), a nie tylko te z końca.

Nazwane parametry

```
def func (a, b, c = 5, d)
{
    a * b + c * d;
}
func (1, 2, d = 3, c = 4);
func (1, 2, d = 3)
```

Krotki

- Tworzymy je oddzielając elementy przecinkami.
- Możemy je indeksować jak tablice, ale jedynie stałymi.
- Do dekonstrukcji krotki można użyć instrukcji wielokrotnego przypisania.
- Jeśli podamy krotkę funkcji oczekującej k argumentów, zostanie ona automatycznie rozbita.

Krotki

```
def f (x1, x2, x3) {
    System.Console.WriteLine ($"$x1$x2 = $x3");
};
def tuple = ("x", 2, 3);
def x1 = tuple [0];
def (_, x2, x3) = tuple;
f (tuple)
```

Listy

- Listę tworzy się operatorem `::`.
- Można też podać w kwadratowych nawiasach elementy (oddzielone przecinkami).
- Listy nie można zmieniać (ale można zamiast list użyć `ArrayList`.)

Listy

```
using Nemerle.Collections;
using System;
def l1 = [4, 5, 6, 7];
def l2 = 1 :: 2 :: 3 :: l1;
Console.WriteLine (List.Filter (l2, _ > 3))
```

Warianty

- Warianty są klasami - czyli są przekazywane przez referencję.
- Można też definiować w nich metody.

Warianty

```
variant Color {
    | White
    | Black
    | Other {
        r : int; g : int; b : int;
    }
}

variant Number {
    | FortyTwo {
        public override GetValue () : int { 42 }
    }
    | Other {
        val : int;
        public override GetValue () : int { val }
    }
    public abstract GetValue () : int;
}

using Number;

_ = Color.White ();
def x = FortyTwo ();
System.Console.WriteLine (x.GetValue ())
```

Podstawy

- Korzystamy ze słowa kluczowego `match`.
- Możemy pogrupować kilka wzorców o tej samej akcji.
- Możemy opuścić `match` jeżeli wyrażenie jest całą treścią funkcji.

Dopasowanie wzorca

```
public module Mod {
    public describe (x : list [int]) : string
    {
        match (x) {
            | [] => "Empty list"
            | [x] => $"Single element:  $x"
            | [x,y] with z = 0
            | [x,y,z] => $"List with 2 or 3 elements ($x,$y,?$z)"
            | x::_ => $"Long list with head $x"
        }
    }
    public makesSense (_ : Number) : bool {
        | FortyTwo ()
        | Other (42)  => true
        | _ => false
    }
}
```

Nemerle

Tadeusz Sznuk

Rodzaje wzorców

- Stałe (42, []).
- Zmienne (używane w akcjach).
- _ (dowolna wartość).
- Konstruktory typu wariantowego.
- Rekordy.
- Sprawdzenie typu (`is`).

Rodzaje wzorców

- as (związanie fragmentu wzorca).
- `with`.
- Wskazanie typu (czasem kompilator potrzebuje podpowiedzi.)

Przykłady wzorców

```
variant V {
    | A {x : list [int]}
    | B {y : int};
}
def f1 (_) {
    | V.A([42])
    | V.B (42) => "cool"
    | V.A([]) with x = 0 | V.A([x]) | V.B (x) => $"":  $x"
    | V.A (_) => "A (...)"
}
def f2 (_ : System.Object) {
    | x is int => x
    | x is string => System.Convert.ToInt32 (x)
    | _ => -1
}
System.Console.WriteLine (f2 ("42"));
System.Console.WriteLine (f2 (42));
```

Wyrażenia regularne

Pakiet `Nemerle.Text` umożliwia wykonanie `match` na wyrażeniach regularnych.

regexp match

```
using Nemerle.Text;

System.Console.WriteLine (
regexp match (System.Console.ReadLine ()) {
  |"a+.*"=> "a";
  |@"(?<num : int>\d+)-\w+"=>  $" $(num + 3) ";
  |"(?<name>(Ala|Kasia))? ma kota"=>
     match (name) {
       |Some (n) => $" $n "
       |None => " ? "
     }
  |_=> "?????"
}

)
```

Wcięcia

- W Nemerle białe znaki nie mają znaczenia
- Chyba, że podamy przy kompilacji opcję `-i`
- Albo wpiszemy przed kodem `#pragma indent`
- Wtedy otrzymamy składnię sterowaną wcięciami, podobnie jak w Haskell´u.
- Wewnątrz nawiasów (,), {, }, [,] obowiązuje klasyczna składnia.

System typów

- System typów jest podobny do C#.
- W szczególności obsługuje klasy/metody polimorficzne.
- Tylko że zamiast < używa się [.
- W przykładach przewinął się już jeden taki typ `list`[int]
- ...

Problemy

- Jaki typ należy nadać funkcji `fun (x, y) { x.doSomething (y) }`
- Uwzględniając fakt, że wiele klas może zawierać metodę `doSomething`?
- A nawet kilka takich metod o różnych sygnaturach (`doSomething(string)`, `doSomething(object)`) ?
- I że typu "klasy–które–mają–metodę–X" nie przewiduje się? (i tak nie rozwiązałoby to problemu z przeciążaniem).

Pomysł

- Zwykle funkcje definiujemy po to, żeby ich gdzieś użyć.
- Można więc poczekać z typowaniem, aż się więcej o funkcji dowiemy.
  - Co nie jest wcale takie proste.
  - Utrudnia wygenerowanie sensownego komunikatu w razie błędu.
  - W trakcie typowania mamy niepełną informację.
- Zauważmy, że tylko typy lokalnych zmiennych są zgadywane - pozostałe wciąż trzeba deklarować.

Pomysł cd.

- Z wystąpień funkcji w kodzie powstaje układ równań na typach.
- Można go rozwiązywać od razu po napotakniu wystąpienia.
- Albo zebrać wszystkie dane i dopiero potem szukać rozwiązania.
- Drugie podejście da bardziej "domyślny" kompilator, ale w razie błędu zupełnie nie ma skąd wziąć dobrego komunikatu . . .

Pomysł 2

- Można zrobić coś pośredniego - spróbować wykorzystać jak najwięcej informacji od razu po jej uzyskaniu, ale resztę pracy zostawić na później
- Zainteresowanych szczegółami zapraszam do przejrzenia materiałów z `http://Nemerle.org` i licznych prezentacji z `http://nemerle.org/svn/`.

Edytory

- Ze strony projektu można sciągnąć pliki pozwalające podświetlać składnię w większości popularnych edytorów (Vim, Emacs, MC, . . .).
- Istnieje wtyczka do SharpDevelop 1.0, wersja dla 2.0 jest tworzona.
- MonoDevelop zawiera obsługę Nemerle w swojej podstawowej dystrybucji.
- Wtyczka dająca pełną obsługę Nemerle w Visual Studio jest tworzona, obecne wersje są już nawet trochę stabilne (bardziej niż Orcas).

Cs2n

Cs2n jest narzędziem umożliwiającym konwersję kodu z C# do Nemerle.

- Daje dobry argument w dyskusjach z fanami C#.
- W świetle łatwej współpracy między C# a Nemerle ma charakter raczej ideologiczno-artystyczny niż praktyczny.

NAnt

Proces kompilacji można zautomatyzować przy pomocy narzędzia NAnt. Odpowiednie zadanie jest zawarte w dystrybucji Nemerle.

NAnt

```
<?xml version="1.0"?>
<project name="Sudoku" default="build">
    <include buildfile="properties.xml" />
...
    <target name="build" depends="resources.build">
        <mkdir dir="${bindir}" />
        <mkdir dir="${docdir}" />
        <ncc target="winexe" output="${bindir}\${basename}.exe" debug="true">
            <arg line="-doc ${docdir}\docs.xml" />
            <sources>
                <include name="${srcdir}\*.n"/>
            </sources>
            <references>
                <include name="System.Windows.Forms.dll" />
                <include name="System.Drawing.dll" />
            </references>
        </ncc>
    </target>

</project>
```

NemerleUnit

NemerleUnit to zestaw makr ułatwiających tworzenie testów w NUnit. Oczywiście ww. testy można tworzyć również bez tego pakietu. NemerleUnit nie działa z obecną stabilną wersją Nemerle (0.9.3) - trzeba użyć SVN.

Przykładowe testy

```
#pragma indent
using NemerleUnit
setup
    def abc = [“a”,“b”,“c”]
teardown
    ()
test “length of an empty list”
    assert [].Length equals 0
test “length of a three element list”
    assert abc.Length equals 3
test “equals”
    assert abc equals [“a”,“b”,“c”]
    assert abc does not equal [“c”,“b”,“a”]
test “contains”
    assert abc.Contains(“a”)
    assert abc.Contains(“b”)
    assert abc.Contains(“c”)
```

NDoc/NUnit/NCover

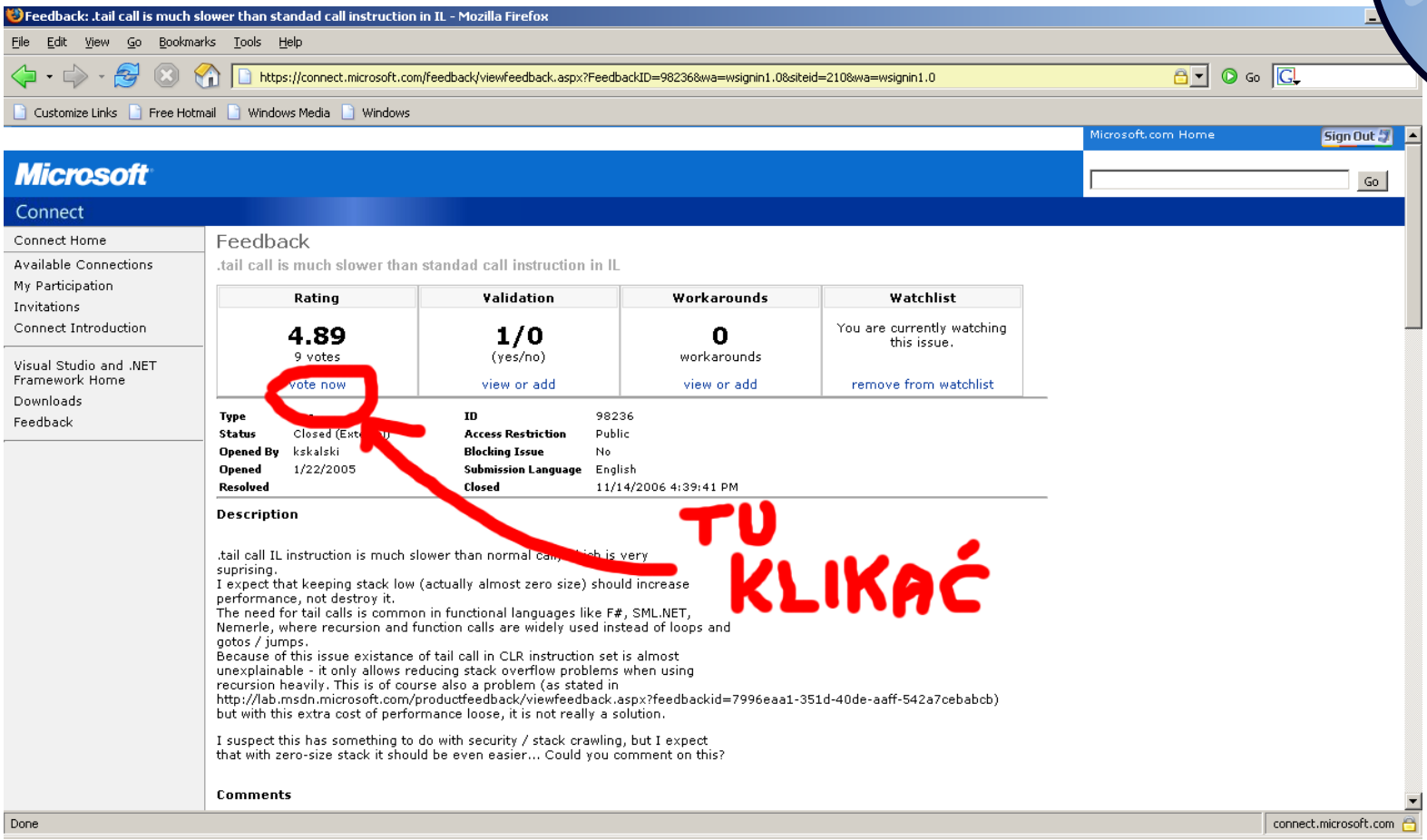
W zasadzie narzędzi tych można używać z Nemerle, ale ...

- NDoc w swojej podstawowej wersji nie zawiera obsługi dla typów polimorficznych.
- Żadna jego wersja nie przedstawi dobrze takich elementów jak np. warianty.
- Nemerle tworzy pewne pomocnicze pola, metody i klasy. Mogą one sprawiać pewne kłopoty przy korzystaniu z narzędzi typu NCover.
- Testy w NUnit pisze się dość wygodnie, zwłaszacza przy wsparciu makr z NemerleUnit.

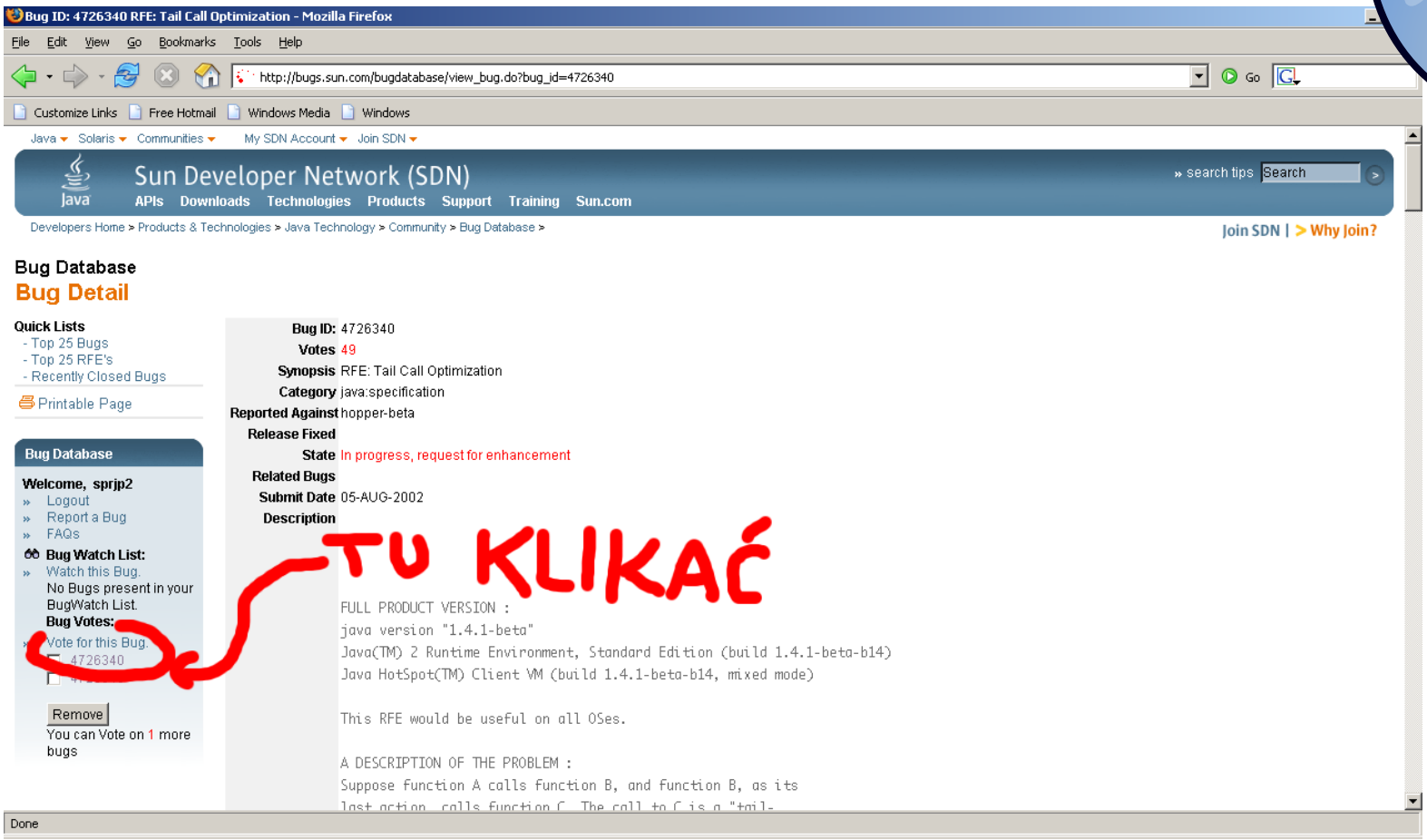
Wywołania ogonowe w wersji M$

- Maszyna wirtualna `.NET` zawiera instrukcję wywołania ogonowego.
- Czyli jest lepiej niż w Javie.
- Niestety, wywołanie ogonowe jest *wielokrotnie* wolniejsze niż zwykłe.
- ???
- Bug ID = 98236 (Java = 4726340)

Prośba I


Feedback: .tail call is much slower than standad call instruction in IL - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
https://connect.microsoft.com/feedback/viewfeedback.aspx?FeedbackID=98236&wa=wsignin1.0&siteid=210&wa=wsignin1.0
Customize Links Free Hotmail Windows Media Windows
Microsoft.com Home
Sign Out
Microsoft
Connect
Go
Connect Home
Available Connections
My Participation
Invitations
Connect Introduction
Visual Studio and .NET Framework Home
Downloads
Feedback
Feedback
.tail call is much slower than standad call instruction in IL
Rating
4.89
9 votes
vote now
Validation
1/0
(yes/no)
view or add
Workarounds
0
workarounds
view or add
Watchlist
You are currently watching this issue.
remove from watchlist
Type
Status
Closed (Ext
Opened By
kskalski
Opened
1/22/2005
Resolved
ID
98236
Access Restriction
Public
Blocking Issue
No
Submission Language
English
Closed
11/14/2006 4:39:41 PM
Description
.tail call IL instruction is much slower than normal ca
suprising.
I expect that keeping stack low (actually almost zero size) should increase performance, not destroy it.
The need for tail calls is common in functional languages like F#, SML.NET, Nemerle, where recursion and function calls are widely used instead of loops and gotos / jumps.
Because of this issue existance of tail call in CLR instruction set is almost unexplainable - it only allows reducing stack overflow problems when using recursion heavily. This is of course also a problem (as stated in http://lab.msdn.microsoft.com/productfeedback/viewfeedback.aspx?feedbackid=7996eaa1-351d-40de-aaff-542a7cebabcb) but with this extra cost of performance loose, it is not really a solution.
I suspect this has something to do with security / stack crawling, but I expect that with zero-size stack it should be even easier... Could you comment on this?
Comments
TU KLIKAĆ
Done
connect.microsoft.com

Prośba II


Bug ID: 4726340 RFE: Tail Call Optimization - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4726340
Customize Links Free Hotmail Windows Media Windows
Java Solaris Communities My SDN Account Join SDN
Sun Developer Network (SDN)
APIs Downloads Technologies Products Support Training Sun.com
» search tips Search
Developers Home > Products & Technologies > Java Technology > Community > Bug Database >
Join SDN | > Why Join?
Bug Database
Bug Detail
Quick Lists
- Top 25 Bugs
- Top 25 RFE's
- Recently Closed Bugs
Printable Page
Bug Database
Welcome, sprjp2
» Logout
» Report a Bug
» FAQs
Bug Watch List:
» Watch this Bug.
No Bugs present in your BugWatch List.
Bug Votes:
» Vote for this Bug.
4726340
Remove
You can Vote on 1 more bugs
Bug ID: 4726340
Votes 49
Synopsis RFE: Tail Call Optimization
Category java:specification
Reported Against hopper-beta
Release Fixed
State In progress, request for enhancement
Related Bugs
Submit Date 05-AUG-2002
Description
TU KLIKAĆ
FULL PRODUCT VERSION :
java version "1.4.1-beta"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1-beta-b14)
Java HotSpot(TM) Client VM (build 1.4.1-beta-b14, mixed mode)
This RFE would be useful on all OSes.
A DESCRIPTION OF THE PROBLEM :
Suppose function A calls function B, and function B, as its
Done

Część III: Metaprogramowanie

7. Podstawy

8. Zastosowania
   - Makro $
   - Wzorce projektowe
   - Makro `late`
   - `SQL`
   - ...

9. Tworzenie makr
   - Proste makro
   - Drzewa składniowe
   - Rozszerzanie składni
   - Fazy kompilacji
   - Definiowanie typów
   - α-konwersja

Nemerle

Tadeusz Sznuk

Makra

Makro jest funkcją, która wykonywana jest podczas kompilacji programu. Makra stosuje się na różne sposoby

- Jak funkcję - zwykle zwracają wtedy wyrażenie, które wstawiane jest w kod w miejscu wywołania makra.
- Jak atrybut (np. Accessor) - mogą wówczas wproawdzić róznorakie modyfikacje (dodać nowe pole, nową klasę, ...).
- Niektóre makra rozszerzają składnię Nemerle.

Makro $

Pozwala wstawić w treść napisu proste wyrażenia. Czasem nie radzi sobie z bardziej złożonymi.

Przykład

```
def x = 21;
def y = 21;
System.Console.WriteLine ($"$x + $y = $(x + y)")
```

Nemerle

Tadeusz Sznuk

Pośrednik

Chwilowo jednyne tego typu makro w bibliotece standardowej.

Przykład

```
using Nemerle.DesignPatterns;

public interface Cool {
    GetX () : int;
};

public class X {
    [Nemerle.DesignPatterns.ProxyPublicMembers ()]
    c : Cool;
    public this (x : Cool) { c = x }
};
def f (x : X) : int {
    x.GetX ()
}
```

Makro late

Makro to umożliwia korzystanie z mechanizmu dynamicznych wywołań. Tylko po co?

Przykład

```
using Nemerle.Late;

public class X {
    public virtual Length  : int { get {41} };
};
public class Y : X {
    public override Length  : int { get {42} };
};
def getLength (obj : object) {
  late obj.Length
}
def f (x : X) {
    System.Console.WriteLine (getLength (x));
};
System.Console.WriteLine (getLength ("Nemerle"));
System.Console.WriteLine (getLength ([1,2,3]));
f (Y ())
```

Makra SQL

Nemerle zawiera zestaw makr ułatwiających korzystanie z bazy danych. Zapytania są wysyłane do bazy *w czasie kompilacji* celem sprawdzenia ich poprawności i odkrycia struktury wyników.

SQL - przykład

Przykład

```
[assembly: ConfigureConnection ("Npgsql.NpgsqlConnection",
"Server=localhost;Database=test;"
                    "User ID=postgres;Password=sql;")]
public class Test
{
    static insert (conn : NpgsqlConnection, x : string, y : string) : void
    {
      _ = ExecuteNonQuery ("INSERT INTO employee VALUES ($x, $y)", conn);
    }

    public static Main() : void
    {
       def connectionString =
          "Server=localhost;"+
          "Database=test;"+
          "User ID=postgres;"+
          "Password=sql;";
       def dbconn = NpgsqlConnection (connectionString);
       dbcon.Open ();
       def myparm = "Ryszard";
       ExecuteReaderLoop (
        "SELECT * FROM employee WHERE firstname = $myparm", dbcon,
       {
         Nemerle.IO.printf ("Name:  %s %s\n", firstname, lastname)
       });
       dbconn.Close();
    }
}
```

Inne zastosowania makr

- Asercje.
- Konstrukcje współbieżne.
- Częściowe wyliczanie funkcji w czasie kompilacji.
- ...

Kod prostego makra

Poniższy przykład ilustruje kod prostego makra, wypisującego komunikat podczas kompilacji i w czasie działania.

Proste makro

```
macro m () {
  Nemerle.IO.printf ("compile-time\n");
  <[ Nemerle.IO.printf ("run-time\n") ]>;
}
```

Kompilacja i użycie

Makro kompilujemy poleceniem

Kompilacja makr

```
ncc -r Nemerle.Compiler.dll -t:dll mymacro.n -o mymacro.dll
```

W kodzie programu możemy użyć tegoż makra

Użycie

```
module M {
  public Main () : void {
    m ();
  }
}
```

o ile nie zapomnimy podać odpowiednich paramatrów przy kompilacji.

Kompilacja

```
ncc -r mymacro.dll myprog.n -o myprog.exe
```

Drzewa składniowe

- Kod ujęty w <[]> traktowany jest jak drzewo składniowe.
- Makro jest funkcją zwracającą takie drzewo.
- Parametry makra również są drzewami składniowymi.
- Notacja <[]> może być użyta do dekompozycji drzew składniowcych.

Drzewa składniowe

Przykład wykorzystania argumentów makra w zwracanym drzewie

Przykład

```
macro for (init, cond, change, body)
{
  <[
    $init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}
```

Użycie makra

```
for (mutable i = 0, i < 10, i++, printf ("%d", i))
```

Nemerle

Tadeusz Sznuk

Stałe

Czasami zamiast drzewa składniowego stałej chcielibyśmy dostać jej wartość. Można to zrobić w sposób zilustrowany w poniższym kodzie.

Przykład

```
using System;
using Nemerle.Compiler.Parsetree;

module MyModule {
  public print_compilation_time () : PExpr
  {
    <[ System.Console.WriteLine ($(DateTime.Now.ToString () : string)) ]>
  }
}
```

Konstrukcje o zmiennej liczbie elementów

Niektóre konstrukcje składniowe mają listę argumentów, która nie ma z góry znanej długości. W drzewie składniowym możemy użyć konstrukcji {.. $exps } by wstawić wszytkie elementy listy.

Przykład

```
mutable exps = [ <[ printf ("%d ", x) ]>,
    <[ printf ("%d ", y) ]> ];
exps = <[ def x = 1 ]> :: <[ def y = 2 ]> :: exps;
<[ {.. $exps } ]>
```

Konstrukcje o zmiennej liczbie elementów

W podobny sposób możemy uzyskać dostęp do elementów danego ciągu wyrażeń.

Przykład

```
using Nemerle.Collections;

macro castedarray (e) {
match (e) {
  |<[ array [.. $elements ] ]> =>
     def casted = List.Map (elements, fun (x) { <[ ($x : object) ]> });
     <[ array [.. $casted] ]>
  |_ => e
}
}
```

Rozszerzanie składni

Makra mogą na różne sposoby rozszerzać składnię Nemerle. Prosty sposób sprowadza się do zdefiniowania słów kluczowych wyznaczających wywołanie makra.

Przykład

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")",
body)
{
  <[
    $init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}
```

Rozszerzanie składni

- Możemy poprosić Nemerle o przekazanie zamiast drzewa składniowego ciągu symboli leksykalnych.
- Ciąg ten będzie już częściowo sparsowany - do postaci drzewa nawiasowego.
- ...
- Zauważmy, że symbole $<$ i $>$ nie są traktowane jak nawiasy.

Makra atrybutopodbne

Niektóre makra stosujemy jak atrybuty dla klas, metod czy pól. Kod takiego makra mógłby wyglądać tak

Makro - atrybut dla metody

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                     Nemerle.MacroTargets.Method)]
macro MethodMacro (t : TypeBuilder, f : MethodBuilder, expr)
{
  // użyj t i f do badania lub zmiany elementów na poziomie klasy
  //
}
```

Fazy kompilacji

W przypadku atrybutopodobnego makra bardzo istotny jest moment, w którym zostanie ona uruchomione. Mamy następujące możliwości

BeforeInheritance Po sparsowaniu programu i uwzględnieniu zadeklarownych typów, ale nie dziedziczenia. Makro może zmieniać hierarchię klas.

BeforeTypedMembers Po uwzględnieniu informacji o dziedziczeniu.

WithTypedMembers Po dokładnym otypowaniu pól i metod, nie można już zmieniać ich sygnatur.

Definiowanie klas w makrach

Z poziomu makra możemy łatwo definiować nowe klasy. Prosty przykład ilustrujący sposób tworzenia i modyfikowania typów jest w poniższym kodzie.

Przykład

```
macro BuildClass ()
{
  def ctx = Nemerle.Macros.ImplicitCTX ();
  def builder = ctx.Env.Define (<[ decl:
    internal class FooBar
    {
      public static SomeMethod () : void
      {
        System.Console.WriteLine (42);
      }
    }
  ]>);

  builder.Compile ();

  <[ FooBar.SomeMethod () ]>
}
```

α-konwersja

- W drzewach zwracanych przez makra występują definicje. Byłoby niedobrze, gdyby przesłaniały one definicje z punktu wywołania makra.
- Symbole w makrach należą do klasy `Name` i zawierają oprócz nazwy informację o miejscu definicji, dzięki czemu unikają konfliktów.
- Można też użyć metody `Macros.NewSymbol()`, która zwróci nieużywany nigdzie symbol.
- Czasem jednak *chcemy* skorzystać z kontekstu wywołania - używamy wtedy `Nemerle.Macros.UseSiteSymbol (name : string) : Name`

Makro singleton

Kod makra

```
using Nemerle.Compiler;
using Nemerle.Collections;

namespace DesignPatterns
{
  [Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
                       Nemerle.MacroTargets.Class)]
  macro Singleton (t : TypeBuilder, getter)
  {
    def mems = t.GetParsedMembers ();
    // find constructor, which we will need to call
    // to create instance
    def ctor = List.Filter (mems, fun (x) {
      |<[ decl: ..$_this (..$_) $_]> => true
      |_ => false
    });
    match (ctor) {
      |[ <[ decl: ..$_this (..$parms) $_]> as constructor ] =>
        match (getter) {
          |<[ $(getter_name : name) ]> =>
            // we must prepare expressions for invoking constructor
            def invoke_parms = List.Map (parms, fun (x) {
              <[ $(x.ParsedName : name) ]>
            });
```

cd.

Kod cd.

```
            // first define the field, where a single instance will be stored
            t.Define (<[ decl:
              private static mutable instance : $(t.ParsedName : name);
            ]>);

            // finally, define getter
            t.Define (<[ decl:
              public static $(getter_name : name) : $(t.ParsedName : name) {
                get {
                  // lazy initialization in generated code
                  when (instance == null)
                    instance = $(t.ParsedName : name) (..$invoke_parms);
                  instance;
                }
              }
            ]>);

            // make sure constructor is protected
            constructor.Attributes |= NemerleAttributes.Protected;

          |_=>
            Message.FatalError ($"Singleton must be supplied with a simple name for
getter, got $getter")
        }
      |_=> Message.Error ("Singleton design pattern requires exactly one constructor
defined")
    }
  }
}
```