PLDIr#5 (PLDI2001)

2010-01-06　酒井 政裕

Citation Count: 43

Design and Implementation of Generics for the .NET Common Language Runtime

Andrew Kennedy (MSR),
Don Syme (MSR)

背景

- JVMや当時のCLRは(パラメトリック)多相=Genericsをサポートしてない
- なので、多相をコンパイルしようとすると、色々問題が……
  - プリミティブ型によるインスタンス化の禁止(GJ, NextGenのerasure変換)
  - “raw type” casting semantics (GJ)
  - 分割コンパイル出来ない (MLj)
  - 複雑なコンパイル方法 (NextGen)
  - プリミティブ型でインスタンス化したときのパフォーマンスの問題 (PolyJ, Pizza)
- なので、ちゃんとCLRレベルで実装しましょう

Objectベースのスタック

```
class Stack {
  private object[]  store;

  private int size;

  public Stack() {
    store = new object[10];
size=0;
  }

  void Push(object item) {
    if (size >= store.Size) {
      object[] tmp = new
object[size*2];
      Array.Copy(store, tmp, size);
      store = tmp;
    }
    store[size++] = x;
  }
```

```
  public object Pop() {
    return store[--size];
  }


  public static void Main() {
    Stack x = new Stack();
    x.Push(17); // boxing
    Console.WriteLine(x.Pop() ==
17)
  }
}
```

Generic C# のスタック

```
class Stack<T> {
  private T[] store;

  private int size;

  public Stack() {
    store = new T[10]; size=0;
  }


  void Push(T item) {
    if (size >= store.Size) {
      T[] tmp = new T[size*2];
      Array.Copy<T>(store, tmp,
size);
      store = tmp;
    }
    store[size++] = x;
  }
```

```
  public T Pop() {
    return store[--size];
  }


  public static void Main() {
    Stack x = new Stack<int>();
    x.Push(17);
    Console.WriteLine(x.Pop() ==
17)
  }

}
```

Design Choice

- クラス、インターフェース、構造体、メソッドはすべて型によるパラメータ化可能
- 厳密な実行時型:
  - e.g. List<string> と List<object> は区別
- インスタンス化は無制限にできる
  - e.g. List<int>, List<double>, …
- F-bounded polymorphism

Design Choice (2)

- インスタンス化されたクラスやインターフェースを継承可能
- 多相再帰(Polymorphic recursion)もOK
  - 多相的なクラスのメソッドは、異なる型でインスタンス化された自分自身を呼び出せる
  - 多相的なメソッドも、異なる型でインスタンス化された自分自身を呼び出せる
- 多相的な仮想関数もOK

Design Choice (3)

- 通常のクラスベースの言語はカバー
- それ以外の色々な言語では、サポートできなかったものも……
  - Haskellの高階型と種 (e.g. List : *→*)
  - ML系言語のモジュールの、高階型を使ったエンコーディング
  - HaksellとMercuryの型クラス
  - Eiffelの、型構築子の型安全でない共変性
  - C++のtemplateも完全にはサポートしない

実装

- 伝統的なコンパイル・リンク・実行というモデルに囚われず、CLRの動的ローディングや実行時コード生成を活用
- ポイント
  - 実行時にコードを特化 (JIT type specialization)
  - コードと表現は可能な限り共有
  - ボックス化しない
  - 実行時型の効率的なサポート

ILの拡張

オブジェクトベースのスタック

```
.class Stack {
  .field private class System.Object[]
store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr System.Object
    stfld class System.Object[]
Stack::store
    ldarg.0
    ldc.i4 0
    stfld int32 Stack::size
    ret
  }
```

多相的なスタック

```
.class Stack<T> {
  .field private !0[] store

  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr !0
    stfld !0[] Stack<!0>::store

    ldarg.0
    ldc.i4 0
    stfld int32 Stack<!0>::size
    ret
  }
```

ILの拡張

オブジェクトベースのスタック

```
.class Stack {
  .field private class System.Object[]
store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr System.Object
    stfld class System.Object[]
Stack::store
    ldarg.0
    ldc.i
    stfl
    ret
  }
```

多相的なスタック

```
.class Stack<T> {
  .field private !0[] store

  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr !0
    stfld !0[] Stack<!0>::store

    ldarg.0
```

型パラメータ情報の追加

ldargなどの命令は、もともと複数の型に適用できる(JITが型を自動的に決定して型に応じたコード生成)ので、そのまま。

ILの拡張

```
.method public void Push(class
System.Object x) {
  .maxstack 4
  .locals (class System.Object[], int32)
  …
  ldarg.0
  ldfld class System.Object[] Stack::store
  ldarg.0
  dup
  ldfld int32 Stack::size
  dup
  stloc.1
  ldc.i4 1
  add
  stfld int32 Stack::size
  ldloc.1
  ldarg.1
  stelem.ref
  ret
}
```

```
.method public void Push(!0 x) {

  .maxstack 4
  .locals (!0[], int32)
  …
  ldarg.0
  ldfld !0[] Stack<!0>::store
  ldarg.0
  dup
  ldfld int32 Stack<!0>::size
  dup
  stloc.1
  ldc.i4 1
  add
  stfld int32 Stack<!0>::size
  ldloc.1
  ldarg.1
  stelem.any !0
  ret
}
```

stelemは型に依存する命令なので、新たにstelem.anyを追加

ILの拡張

```
.method public static void Main() {
  .entrypoint
  .maxstack 3
  .locals (class Stack)
  newobj void Stack::.ctor()
  stloc.0
  ldloc.0
  ldc.i4 17
  box System.Int32
  call instance void Stack::Push(class
System.Object)
  ldloc.0
  call instance class System.Object
Stack::Pop()
  unbox System.Int32
  ldind.i4
  ldc.i4 17
  ceq
  call void System.Console::WriteLine(bool)
  ret
}
```

```
.method public static void Main() {
  .entrypoint
  .maxstack 3
  .locals (class Stack<int32>)
  newobj void Stack<int32> ::.ctor()
  stloc.0
  ldloc.0
  ldc.i4 17

  call instance void Stack<int32>::Push(!0)
  ldloc.0
  call instance !0 Stack<int32>::Pop()



  ldc.i4 17
  ceq
  call void System.Console::WriteLine(bool)
  ret
}
```

box/unboxが不要に

ILの拡張: 禁止されていること

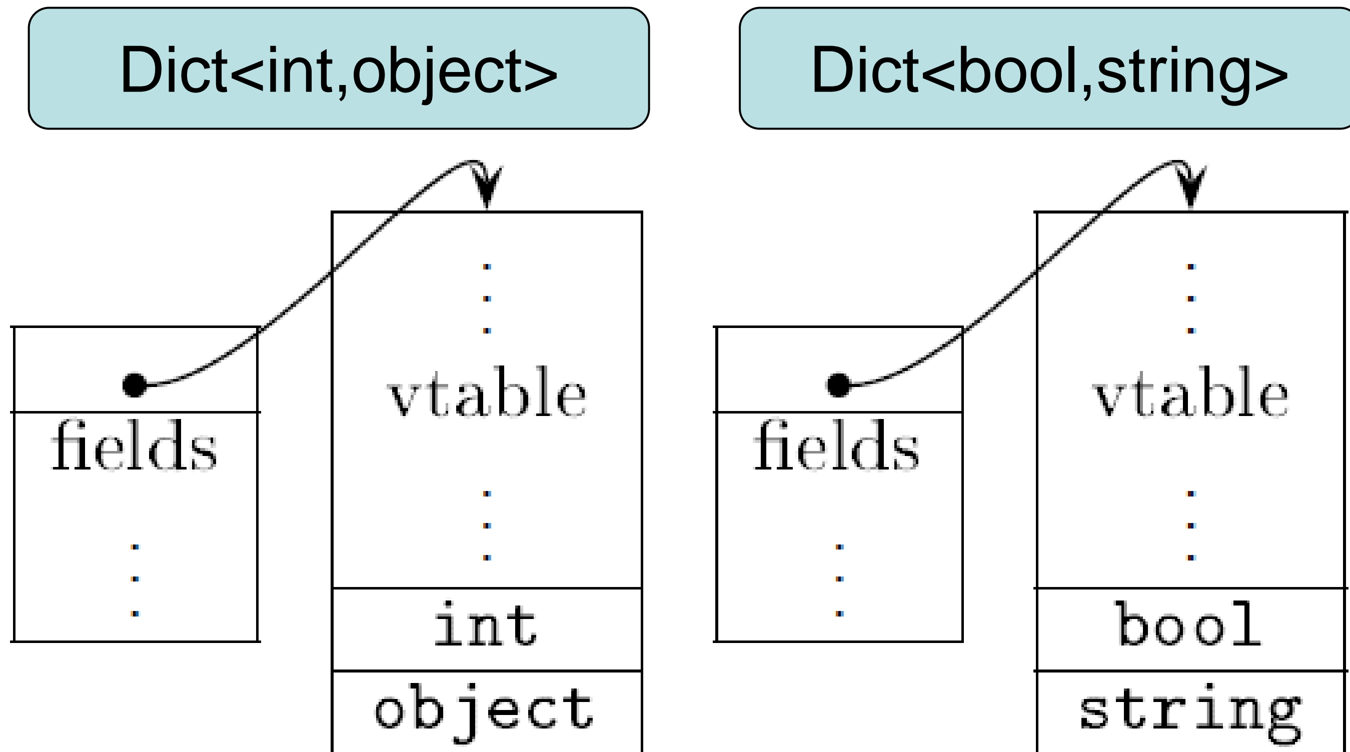
- .class Foo<T> extends !0
- !0::.ctor()
- !0::.myMethod()
- クラスの型パラメータの静的変数での使用
- パラメタ化されたインターフェースをインスタンス化したものを、複数回実装すること

ランタイム: データ表現とコード

- 伝統的には、データ表現とコードを、
  - 型パラメータ毎に特化、
  - もしくは共有
- CLRではこれらをミックス出来る
  - あるインスタンス化が必要になった時点で、互換性のあるインスタンス化がすでにされているか探す
  - なかったら新たに生成
  - 生成されたvtableのエントリは、呼ばれたときに実際にコード生成するスタブ
    - polymorphic recursionなどに対応するため: e.g. **class** C<T> { **void** m() { … **new** C<C<T>>() … } }

ランタイム: 型情報

- 厳密な型情報はvtable内に持つ
(vtable を type handle として使う)



ランタイム: 型パラメタへのアクセス

```
class C { virtual void m(int) { … } }
class D<T> : C {
  void p(Object x) { … (Set<T[]>) x … }
  override void m(int) { … new List<T> … }
}
class E<U> : D<Tree<U>> {
  // m inherited
  void q() { … y is U[] … }
}
```

毎回、型情報をlookupするのは大変なので、dictionaryをlazyに生成して……

ランタイム: Eのvtable内のdictionary

```
class C { virtual void
   m(int) { … } }
class D<T> : C {
  void p(Object x) { …
   (Set<T[]>) x … }
  override void m(int)
   { … new List<T> … }
}
```

```
class E<U> :
   D<Tree<U>> {
  // m inherited
  void q() { … y is
   U[] … }
}
```

Class	Slot no.	Type parameter or open type
D	0	T = Tree<U>
D	1	Set<T[]> = Set<Tree<U>[]>
D	2	List<T> = List<Tree<U>>
E	3	U
E	4	U[]

パフォーマンス

S := new stack
c := constant
for m ∈ 1···10000 do
  S.push(c) m times
  S.pop() m times

Element type	Time (seconds)		
	Object	Poly	Mono
`object`	2.9	2.9	2.9
`string`	3.5	2.9	3.1
`int`	8.5	1.8	2.0
`double`	10.4	2.0	2.0
`Point`	10.5	4.3	4.3

パフォーマンス2

- 実行時の型操作のオーバーヘッドを測定
- こんなやつ?
  - class C<T> {
    Main() {
    for … { new Dict<T[], List<T>>() }
    } }

Type	Time (seconds)		
	Specialized	Runtime look-up	Lazy Dictionary
`List<T>`	4.2	288	4.9
`Dict<T[],List<T>>`	4.2	447	4.9

Citation Count: 92

Automatic Predicate Abstraction of C Programs

Thomas Ball (MSR),
Rupak Majumdar (U.C.Berkeley),
Todd Millstein (Univ. of Washinton),
Sriram K. Rajamani (MSR)

- ソフトウェアのモデル検査では抽象化が重要
  - 状態爆発を避けるため
  - そもそも無限状態の場合には、有限状態にする必要がある
- この論文はSLAMが抽象化に使っているC2BPの詳しい話
  - CプログラムPと述語の集合Eから、
    述語抽象化した Boolean program B(P,E) を生成

B(P, E)

- 変数はEの要素
- 元のプログラムの保守的な抽象化
  - 元のプログラムでfeasibleなパスは、抽象化後もfeasible

例: 元のCプログラム P

```
typedef struct cell {
  int val;
  struct cell* next;
} *list;

list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;
  curr = *l;
  prev = NULL;
  newl = NULL;
  while (curr != NULL) {
    nextCurr = curr->next;
    if (curr->val > v) {
      if (prev != NULL) {
        prev->next = nextCurr;
      }
      if (curr == *l) {
        *l = nextCurr;
      }
      curr->next = newl;
L:    newl = curr;
    } else {
      prev = curr;
    }
    curr = nextCurr;
  }
  return newl;
}
```

例: 述語の集合E

- curr == NULL
- prev == NULL
- curr->val > v
- prev->val > v

例: BP(P, E)

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};
  {curr==NULL} = unknown();
  {curr->val>v} = unknown();
  {prev==NULL} = true;
  {prev->val>v} = unknown();
  skip;
  while(*) {
    assume(!{curr==NULL});
    skip;
    if (*) {
      assume({curr->val>v});
      if (*) {
        assume(!{prev==NULL});
        skip;
      }
```

```
      if (*) {
        skip;
      }
      skip;
L:    skip;
    } else {
      assume(!{curr->val>v});
      {prev==NULL} = {curr==NULL};
      {prev->val>v} = {curr->val>v};
    }
    {curr==NULL} = unknown();
    {curr->val>v} = unknown();
  }
  assume({curr==NULL});
}
```

- 自明な話だと思ってたけど、色々工夫が必要
- 元のプログラムのコードに対応して、どの述語の真偽がどう変化するかの判定
  - 最終的には定理証明器を駆動
    - 定理証明器は重いので、工夫して呼ぶ回数を削減
  - ポインタ解析による精度の改善
    - Our implementation uses Das's points-to algorithm [12] to obtain flow-insensitive, context-insensitive may-alias information.

Citation Count: 27

The Pointer Assertion Logic Engine

Anders Møller
& Michael I. Schwartzbach
(University of Aarhus, Denmark)

概要

- ヒープの構造に関する性質を、Pointer Assertion Logic というロジックで、アノテーション(事前条件・事後条件・不変条件)
- プログラムとアノテーションを、WS2S(weak monadic second-order theory of 2 successors)に変換して、MONAで検証する、
  - 条件を満たさない場合には、反例を出力

具体例

```
type List = {data next:List;}

pred roots(pointer x,y:List, set
     R:List)
     = allpos p of List:
          p in R
          <=>
          x<next*>p | y<next*>p;
```

```
proc reverse(data list:List):List
  set R:List;
[roots(list,null,R)]
{
  data res:List;
  pointer temp:List;
  res = null;
  while [roots(list,res,R)] (list!=null)
  {
    temp = list.next;
    list.next = res;
    res = list;
    list = temp;
  }
  return res;
}
[roots(return,null,R)]
```

性能

Example name	Lines of code	Invariants (formulas)	GTA operations	Largest GTA		Time (seconds)	Memory (MB)
				States	BDD nodes		
reverse	16	1	1,109	35	142	0.52	2
search	12	1	853	27	85	0.25	2
zip	33	1	1,753	174	730	4.58	11
delete	22	0	973	73	349	1.36	5
insert	33	0	1,005	103	443	2.66	7
rotate	11	0	590	44	213	0.22	1
concat	24	0	1,056	48	177	0.47	3
bubblesort_simple	43	1	1,477	373	3,289	2.86	18
bubblesort_boolean	43	2	1,737	357	3,922	3.37	12
bubblesort_full	43	2	2,069	373	3,291	4.13	19
orderedreverse	24	1	1,091	29	100	0.46	3
recreverse	15	2	1,019	42	176	0.34	2
doublylinked	72	1	4,163	230	796	9.43	13
leftrotate	30	0	1,489	165	1,550	4.62	7
rightrotate	30	0	1,489	165	1,550	4.68	7
treeinsert	36	1	1,989	137	844	8.27	31
redblackinsert	57	7	4,279	297	2,419	35.04	44
threaded	54	4	3,505	50	248	3.38	7

Figure 1: Statistics from PALE experiments.

※ GTA = Guided Tree Automata

- PAL(Pointer Assertion Logic)は、表現力と検証コストの良いバランス

- 主なターゲットはデータ構造
- アノテーションを書くのは大変だけど、データ構造の設計は元々invariantを注意深く考える必要があるから、それを考えれば大したことない