Subject: Little experiment compiling to function pointer calls...
Posted by mirek on Wed, 16 Nov 2011 18:54:46 GMT
View Forum Message <> Reply to Message

Thinking about TCC and runtime compilation frameworks (also web templates) I have got an idea how represent the code structure in the tree of virtual objects (basically, via function pointers).

I have put together a litte experimental snippet:

```
#include <Core/Core.h>

using namespace Upp;

struct Oper {
 virtual double Execute() = 0;
 virtual ~Oper() {}
};

struct BinOper : Oper {
 One<Oper> a;
 One<Oper> b;
};

struct Add : BinOper {
 virtual double Execute() { return a->Execute() + b->Execute(); }
};

struct Sub : BinOper {
 virtual double Execute() { return a->Execute() - b->Execute(); }
};

struct Mul : BinOper {
 virtual double Execute() { return a->Execute() * b->Execute(); }
};

struct Div : BinOper {
 virtual double Execute() { return a->Execute() / b->Execute(); }
};

struct Const : Oper {
 double value;
 virtual double Execute() { return value; }
};

struct Var : Oper {
 double *var;
 virtual double Execute() { return *var; }
```

```
};

struct Compiler {
 VectorMap<String, double *> var;
 One<Oper> Term(CParser& p);
 One<Oper> Exp(CParser& p);
 One<Oper> Factor(CParser& p);
};

One<Oper> Compiler::Term(CParser& p)
{
 One<Oper> result;
 if(p.IsId()) {
  double *v = var.Get(p.ReadId(), NULL);
  if(!v)
   p.ThrowError("unknown variable");
  result.Create<Var>().var = v;
 }
 else
 if(p.Char('(')) {
  result = Exp(p);
  p.PassChar(')');
 }
 else
  result.Create<Const>().value = p.ReadDouble();
 return result;
}

One<Oper> Compiler::Factor(CParser& p)
{
 One<Oper> result = Term(p);
 for(;;)
  if(p.Char('*')) {
   One<Oper> rr;
   Mul& m = rr.Create<Mul>();
   m.a = result;
   m.b = Term(p);
   result = rr;
  }
  else
  if(p.Char('/')) {
   One<Oper> rr;
   Div& m = rr.Create<Div>();
   m.a = result;
   m.b = Term(p);
   result = rr;
  }
  else
```

```
   return result;
}

One<Oper> Compiler::Exp(CParser& p)
{
 One<Oper> result = Factor(p);
 for(;;)
  if(p.Char('+')) {
   One<Oper> rr;
   Add& m = rr.Create<Add>();
   m.a = result;
   m.b = Factor(p);
   result = rr;
  }
  else
  if(p.Char('-')) {
   One<Oper> rr;
   Sub& m = rr.Create<Sub>();
   m.a = result;
   m.b = Factor(p);
   result = rr;
  }
  else
   return result;
}

VectorMap<String, double> var;

double Exp(CParser& p);

double Term(CParser& p)
{
 if(p.Id("abs")) {
  p.PassChar('(');
  double x = Exp(p);
  p.PassChar(')');
  return fabs(x);
 }
 if(p.IsId())
  return var.Get(p.ReadId(), 0);
 if(p.Char('(')) {
  double x = Exp(p);
  p.PassChar(')');
  return x;
 }
 return p.ReadDouble();
}
```

```
double Mul(CParser& p)
{
 double x = Term(p);
 for(;;)
  if(p.Char('*'))
   x = x * Term(p);
  else
  if(p.Char('/')) {
   double y = Term(p);
   if(y == 0)
    p.ThrowError("Divide by zero");
   x = x / y;
  }
  else
   return x;
}

double Exp(CParser& p)
{
 double x = Mul(p);
 for(;;)
  if(p.Char('+'))
   x = x + Mul(p);
  else
  if(p.Char('-'))
   x = x - Mul(p);
  else
   return x;
}


CONSOLE_APP_MAIN
{
 double x, y;

 Compiler c;
 c.var.Add("x", &x);
 c.var.Add("y", &y);

 CParser p("1 / (1 - x * y + x - y)");
 One<Oper> fn = c.Exp(p);

 x = 5;
 y = 10;

 RDUMP(1 / (1 - x * y + x - y));
 RDUMP(fn->Execute());
```

```
 {
  RTIMING("Interpreted");
  double sum = 0;
  for(x = 0; x < 1; x += 0.001)
   for(y = 0; y < 1; y += 0.001) {
    var.GetAdd("x") = x;
    var.GetAdd("y") = y;
    sum += Exp(CParser("1 / (1 - x * y + x - y)"));
   }
  RDUMP(sum);
 }
 {
  RTIMING("Compiled");
  double sum = 0;
  for(x = 0; x < 1; x += 0.001)
   for(y = 0; y < 1; y += 0.001)
    sum += fn->Execute();
  RDUMP(sum);
 }
 {
  RTIMING("Direct");
  double sum = 0;
  for(x = 0; x < 1; x += 0.001)
   for(y = 0; y < 1; y += 0.001)
    sum += 1 / (1 - x * y + x - y);
  RDUMP(sum);
 }
}
```

This compares three methods to evaluate expression - interpreted, compiled to the virtual tree and 'native' ("compile time compiled").

Results are interesting:

```
TIMING Direct       :  8.00 ms -  8.00 ms ( 8.00 ms / 1 ), min:  8.00 ms, max:  8.00 ms, nesting: 1 - 1
TIMING Compiled     : 29.00 ms - 29.00 ms (29.00 ms / 1 ), min: 29.00 ms, max: 29.00 ms, nesting: 1 - 1
TIMING Interpreted  : 551.00 ms - 551.00 ms (551.00 ms / 1 ), min: 551.00 ms, max: 551.00 ms, nesting: 1 - 1
```

Means 'compiled' expression evaluation is only 3 times slower than native code, while being 20 times faster than interpreted results.

I believe this approach would scale up to complete compiler of e.g. C. The advantage over TCC is

that such code is 100% portable, no need to emit assembler opcodes...

[/code]

Subject: Re: Little experiment compiling to function pointer calls...
Posted by koldo on Fri, 18 Nov 2011 08:13:47 GMT
View Forum Message <> Reply to Message

Hello Mirek

I understand the results of your test but, could you explain a little further this :
Quote:I believe this approach would scale up to complete compiler of e.g. C. The advantage over TCC is that such code is 100% portable, no need to emit assembler opcodes...

Subject: Re: Little experiment compiling to function pointer calls...
Posted by mirek on Fri, 18 Nov 2011 13:51:10 GMT
View Forum Message <> Reply to Message

koldo wrote on Fri, 18 November 2011 03:13Hello Mirek

I understand the results of your test but, could you explain a little further this :
Quote:I believe this approach would scale up to complete compiler of e.g. C. The advantage over TCC is that such code is 100% portable, no need to emit assembler opcodes...

Well, TCC produces CPU opcodes. That is definitely faster, but also CPU dependent and a little bit harder to interface with 'host' platform (in this case U++).

Using this, you can more easily create 'runtime compiler' for most scripting languages, which would be slower than one producing CPU opcodes, but very likely faster than interpreted bytecode. And very easily interfacable with host (see how 'x' and 'y' variables are easily iterfaced).

The only problem I can see is jumps. 'goto' is likely out of question and I am still unsure if 'break' is possible (maybe the one way is to implement it using exceptions).

Mirek

Subject: Re: Little experiment compiling to function pointer calls...
Posted by koldo on Fri, 18 Nov 2011 15:14:30 GMT
View Forum Message <> Reply to Message

Ok

Subject: Re: Little experiment compiling to function pointer calls...
Posted by mirek on Fri, 18 Nov 2011 17:10:45 GMT
View Forum Message <> Reply to Message

I have played with the concept a bit more and got it to be able to compile:

```
{
 sum = 0;
 for(x = 0; x < 1; x = x + step)
  for(y = 0; y < 1; y = y + step)
   sum = sum + 1 / (1 - x * y + x - y);
}
```

Runs 6 times slower than native C++, 10 times faster than interpreted.

Code can be seen in sandbox... (tooks about 300 lines of U++ for this 'JIT' compiler).

Well, nice idea that can perhaps be of some use in the future. I am ending the experiment now