A Virtual Machine Framework for Domain-Specific Languages

David Fick
Supervisor: Prof. B. Watson
Co-supervisor: Prof. D. Kourie
A domain is characterized by a set of related concepts, which need to be understood, before they can be applied to real world problems. For example, Chemistry is a domain in which the concepts of elements, compounds, and their respective reactions, are among those that are to be understood by chemists if they are to be considered experts in the domain.

In essence, a domain, from a developer’s perspective, is simply a home where related concepts can live happily together.
Domain Specific Languages (DSLs)

- **Domain**
  - A set of classes, all connected via relationships of some multiplicity. Classes do not overlap between domains. A domain can be dependant on, or make use of services of other domains (namespaces).

- **DSL**
  - A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to a particular domain.

- **DSL-program**
  - A program, written in the language of the DSL, i.e. the actual DSL program source code.
Why use DSLs?

- An expert already understands his/her domain and just needs a simple means to express solutions
- Intuitive coding style
- Far fewer lines of code than a generic language like, say C++
Techniques and criteria for each technique

- **DSL techniques**
  - Interpreter
  - Hard-coded VM
  - VM Framework

- **Criteria for Evaluation**
  - Scalability
  - Performance
  - Modularity
CASE Tools provide sophisticated IDEs (Integrated Development Environments)

Examples include BridgePoint, Borland Delphi, Borland C++Builder, Borland JBuilder, Microsoft’s Visual C++ and Eclipse.

Visualize design-time structure, UML, state machines, and even visualize runtime structure, such as stack traces, running threads.
**Integrated Development Environment (IDE)**

**Grammar describing development language**
- exp : exp PLUS term
- exp : exp MINUS term
- exp : term
- term : term TIMES fact
- term : term DIV fact
- term : fact
- fact : LPAR exp RPAR
- fact : NUM

**Components, written in some language, eg C#, providing the language semantics**

```csharp
class Add
{
    public Add (DExp env)
    {
        this.env = env;
    }
    public void Execute ()
    {
        double d1,d2;
        d2 = env.stack.pop ();
        d1 = env.stack.pop ();
        env.stack.push(d1+d2);
    }
}
```

**The actual software currently under development**

```csharp
let fun MyFunc (x) =
    let
        n = x
    in
        9 + sum(i) 1..n (2*i)
    end
end
```

**Test scenarios or use cases**
- call MyFunc (30)

**Translation rules to target language/binaries**

```plaintext
double %fun.name% (double %params[0].name%\ %for i=1..params.count-1% ,double %params[i].name%\ %next%)
{
    %for l in locals%  
        double %l.name%;
    %next%
    return %fun.exp%;
}
```
class Add {
    public Add (DExp env)
    {
        this.env = env;
    }
    public void Execute ()
    {
        double d1, d2;
        d2 = env.stack.pop ();
        d1 = env.stack.pop ();
        env.stack.push (d1 + d2);
    }
}

let fun MyFunc (x) =
    let n = x
    in
        9 + sum (i) 1..n (2*i)
    end
in MyFunc (10) end

let MyFunc (x) =
    let n = x
    in
        9 + sum (i) 1..n (2*i)
    end
end

exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM

Test scenarios or use cases

call MyFunc (30)

Integrated Development Environment

Grammar describing development language

exp : exp PLUS term
exp : exp MINUS term
exp : term
term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM

The actual software currently under development

let fun MyFunc (x) =
    let n = x
    in
        9 + sum (i) 1..n (2*i)
    end
in MyFunc (10) end

Translation rules to target language/binary

double %fun.name% (double %params[0].name%
    %for i=1..params.count-1%
    ,double %params[i].name%
    %next%)
{
    %for l in locals%
    double %l.name%;
    %next%
    return %fun.exp%;
}
DSL-Supported IDE

Integrated Development Environment

Translation rules to target language/binary

double %fun.name% (double %params[0].name%\n%for i=1..params.count-1% ,double %params[i].name%\n%next%)  
{ %for l in locals%  
double %l.name%;  
%next%  
return %fun.exp%;  
}

class Add  
{  
public Add (DExp env)  
{  
this.env = env;  
}  
public void Execute ()  
{  
double d1,d2;  
d2 = env.stack.pop ();  
d1 = env.stack.pop ();  
env.stack.push(d1+d2);  
}  
}

The actual software currently under development

let  
fun MyFunc (x) =  
let  
n = x  
in  
9 + sum(i) 1..n (2*i)  
end  
in  
MyFunc (10)  
end

exp : exp PLUS term  
exp : exp MINUS term  
exp : term  
term : term TIMES fact  
term : term DIV fact  
term : fact  
fact : LPAR exp RPAR  
fact : NUM

Grammar describing development language

Test scenarios or use cases

call MyFunc (30)

Components, written in some language, e.g. C#, providing the language semantics
Grammar describing development language:

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

Components, written in some language, e.g., C#, providing the language semantics:

```java
class Add {
    public Add (DExp env) {
        this.env = env;
    }
    public void Execute () {
        double d1, d2;
        d2 = env.stack.pop();
        d1 = env.stack.pop();
        env.stack.push(d1+d2);
    }
}
```

Test scenarios or use cases:

```java
call MyFunc (30)
```

The actual software currently under development:

```java
let fun MyFunc (x) = let
    n = x
    in
    9 + sum(i) 1..n (2*i)
end
in
MyFunc (10)
end
```

Translation rules to target language/binary:

```java
double %fun.name% (double %params[0].name%
%for i=1..params.count-1%
, double %params[i].name%
%next%)
{
    %for l in locals%
    double %l.name%;
    %next%
    return %fun.exp%;
}
```

DSL-Supported IDE
Integrated Development Environment

Components, written in some language, e.g. C#, providing the language semantics

class Add
{
    public Add (DExp env)
    {
        this.env = env;
    }
    public void Execute ()
    {
        double d1,d2;
        d2 = env.stack.pop ();
        d1 = env.stack.pop ();
        env.stack.push(d1+d2);
    }
}

Translation rules to target language/binary

double \%fun.name% (double \%params[0].name%\
    \%for i=1..params.count-1%\
    ,double \%params[i].name%\n    \%next%)
{
    \%for l in locals%\
    double \%l.name%;
    \%next%\
    return \%fun.exp%;
}

The actual software currently under development

let
    fun MyFunc (x) =
    let
        n = x
    in
        9 + sum(i) 1..n (2*i)
    end
end

Grammar describing development language

exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM

Test scenarios or use cases

call MyFunc (30)

DLL, EXE, Binary

DSL-Supported IDE
Integrated Development Environment

**Grammar describing development language**

```
exp : exp PLUS term
exp : exp MINUS term
exp : term

term : term TIMES fact
term : term DIV fact
term : fact

fact : LPAR exp RPAR
fact : NUM
```

**Components, written in some language, eg C#, providing the language semantics**

```
class Add
{
    public Add (DExp env)
    {
        this.env = env;
    }
    public void Execute ()
    {
        double d1,d2;
        d2 = env.stack.pop ();
        d1 = env.stack.pop ();
        env.stack.push(d1+d2);
    }
}
```

**Test scenarios or use cases**

```
call MyFunc (30)
```

**The actual software currently under development**

```
let
    fun MyFunc (x) =
    let
        n = x
    in
        9 + sum(i) 1..n (2*i)
    end
in
    MyFunc (10)
end
```

**Translation rules to target language/binary**

```
double %fun.name% (double %params[0].name%
%for i=1..params.count-1%
  ,double %params[i].name%
%next%)
{
%for l in locals%
    double %l.name%;
%next%
    return %fun.exp%;
}
```
A DSL Development IDE

- Featuring a VM as part of the CASE Software, and has 2 parts:
  - Custom build a DSL, with syntax and semantics, and allow linking into a namespace written in a platform such as the .NET CLR.
  - The second part will allow the developer to write programs using the DSL that has been specifically customized for his or her particular need.
Develop a DSL for a specific problem domain

Grammar describing DSL, and translation rules (compiler generator)

```
exp : exp1 PLUS term: t1, exp2
exp : term: t1, exp2

term : term: t1 TIMES fact: f1, term2
term : NUM: n, term2

translate exp1
{ translate exp1.e1
  translate exp1.t1
  emit "Add"
}

... translate term2
{ emit "Push 0", term2.n }
```

Use the exported DSL compiler and VM to develop a solution expressed in the DSL

The actual software currently under development

```
let
  fun MyFunc (x) =
    let
      n = x
    in
      9 + sum(i) 1..n (2*i)
    end
  in MyFunc (10) end
```

Test scenarios or use cases

```
call MyFunc (30)
```

The actual software currently under development

```
let
  fun MyFunc (x) =
    let
      n = x
    in
      9 + sum(i) 1..n (2*i)
    end
  in MyFunc (10) end
```

Instructions

- Environments
- Domain

Components, written in some language, e.g. C#, providing the language semantics

Develop a DSL for a specific problem domain

Grammar describing DSL, and translation rules (compiler generator)

```
exp : exp1 PLUS term: t1, exp2
exp : term: t1, exp2

term : term: t1 TIMES fact: f1, term2
term : NUM: n, term2

translate exp1
{ translate exp1.e1
  translate exp1.t1
  emit "Add"
}

... translate term2
{ emit "Push 0", term2.n }
```

Use the exported DSL compiler and VM to develop a solution expressed in the DSL

The actual software currently under development

```
let
  fun MyFunc (x) =
    let
      n = x
    in
      9 + sum(i) 1..n (2*i)
    end
  in MyFunc (10) end
```

Test scenarios or use cases

```
call MyFunc (30)
```
Some example DSLs

- **EXP** – a simple functional expression language
- **RT** – a ray-tracer scene description language
- **LQ** – Little Quilt. A functional language of quilts
Conceptual view of a DSL interpreter

Source DSL

```plaintext
scene
{
    sphere 0.5
    {
        translate 3 3 0
        specular 0 0 1
    }
}
```

**Statically Interpret DSL**
The Domain of RT

vector3
- x, y, z: double
- normalize()
- cross()
- dot()

Ray
- x1, y1, z1: double
- x2, y2, z2: double
- reflect()
- refract()

Scene
- render()
- ShootRay()

RObject
- Intersect(): bool

Quadric
- a1, a2, a3: double
- a22, a23, a33: double
- b1, b2, b3: double
- c: double
- tx, ty, tz: double

\[ q(x, y, z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + 2a_{12}xy + b_1 x + b_2 y + b_3 z + c = 0 \]
RT Example

scene {
  window 300.0 200.0  light -3.5 0.5 2.0
  cop 0.0 4.0 1.0  lookat 0.0 0.0 0.0
  viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

  sphere 0.5 {
    translate 1.0 0.0 0.0  reflective 0.1 0.1 0.1
    specular 0.7 0.0 0.0  diffuse 0.7 0.0 0.0
  }

  sphere 0.5 {
    translate 0.0 -1.0 0.2  reflective 0.2 0.3 0.2
    specular 0.0 0.7 0.0  diffuse 0.0 0.7 0.0
  }

  sphere 0.5 {
    translate -1.0 -0.5 0.0  reflective 0.1 0.1 0.1
    specular 0.0 0.0 0.7  diffuse 0.0 0.0 0.7
  }

  plane -0.1 0.0 1.0 1.0 {
    specular 0.0 0.0 0.0  diffuse 1.0 0.5 0.0
    reflective 0.25 0.25 0.25
  }
}

Adding the ambient Construct

to an Interpreter

- Appending the lexer definition for keyword ambient.

```%
lexer
namespace RtcLexer

%token
"{" %LBRACE
"{" %RBRACE
"scene" %SCENE
...
"viewup" %VIEWUP
"ambient" %AMBIENT
"specular" %SPECULAR
...
```
Adding the ambient Construct to an Interpreter

- Altering the parser for the new ambient construct.

```java
...%
node scene_prop6 : scene_prop
{
    public double d1, d2, d3;
    public scene_prop6 (double d1,double d2, double d3)
    
    {this.d1=d1;this.d2=d2;this.d3=d3;}
}
...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
    %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
```
Adding the ambient Construct to an Interpreter

- Altering the functions for the new ambient construct.

```csharp
public class rtc
{
    public static void Main (string [] argv)
    {
        ...
    }
    ...
    public static void trans_scene_prop (RtcParser.scene_prop node)
    {
        if (node is RtcParser.scene_prop1)
        {
            trans_scene_prop1 ((RtcParser.scene_prop1) node);
        }
        ...
        else if (node is RtcParser.scene_prop6)
        {
            trans_scene_prop6 ((RtcParser.scene_prop6) node);
        }
    }
    ...
    public static void trans_scene_prop6 (RtcParser.scene_prop6 node)
    {
        scene.alterAmbient (node.d1, node.d2, node.d3);  // ambient
        scene.setupAxes ();
    }
    ...
    public static Scene scene;
    public static RObject robject;
    public static Bitmap bitmap;
    public static string filename;
}
DSL Hard-coded Virtual Machines

- Conceptual view of a DSL hard-coded VM

Source DSL

```plaintext
scene
{
    sphere 0.5
    {
        translate 3 3 0
        specular 0 0 1
    }
}
```

Generated Instruction Sequence

- Quad
- Push 1
- Push 0.5
- Push 0.5
- Mul
- Div
- Coeff "all"
DSL Hard-coded Virtual Machines

- Instruction Set Architectures - System ISA and User ISA
System VMs vs Process VMs

- A System VM: Virtualizing software between hardware and OS
System VMs vs Process VMs

- A Process VM: Virtualizing software supports an individual process

Diagram:
- Application Software
- ISA
- Virtualizing Software
  - High-level Language VM (HLL VM)
- Operating System
- Hardware
System VMs vs Process VMs

- Broad classification of VMs

**Process VMs**
- Same ISA
  - Emulators
- Different ISA
  - HLL VMs

**System VMs**
- Same ISA
  - Hosted VMs
- Different ISA
  - Codesigned VMs
**DSL VMs**

- **DSL VM**: A concrete instance of a HLL VM, providing an ISA capable of supporting the semantics of some DSL, i.e. the semantics of a single instruction or combination of some sequence of instructions, will facilitate the semantics of the DSL, and thus will reflect the application domain for which the DSL was designed.

- **DSL-compiled program**: The DSL-program, once it has been compiled into a suitable ordered sequence of instructions ready to be executed on a DSL VM.
DSL VMs

```
scene
{
  sphere 0.5
  {
    translate 3 3 0
    specular 0 0 1
  }
}
```
DSL VMs

- Lexer definition for the RT VM

```plaintext
%lexer
%token
"PUSH" %PUSH
"SPHERE" %SPHERE
"PLANE" %PLANE
"WIN" %WIN
"LIGHT" %LIGHT
"COP" %COP
"LOOK" %LOOK
"VUP" %VUP
"AMB" %AMB
"GEOM" %GEOM
"SPEC" %SPEC
"DIFF" %DIFF
"REFL" %REFL
"REFR" %REFR
"AXES" %AXES
"REN"
[-+]?[0-9]+(\.[0-9]+)? %DOUBLE {val = Double.Parse (yytext);} [ \n\r\f\t]
```
**DSL VMs**

- **Parser definition for the RT VM**

```
asm_list :
asm_list : asm_list asm

asm : PUSH DOUBLE
asm : SPHERE
asm : PLANE
asm : WIN
asm : LIGHT
asm : COP
asm : LOOK
asm : VUP
asm : AMB
asm : GEOM
asm : SPEC
asm : DIFF
asm : REFL
asm : REFR
asm : AXES
asm : REN
```
Adding the ambient Construct to a Hard-coded DSL VM

- Altering the grammar for the new ambient construct on the RT compiler

```java
...%node scene_prop6 : scene_prop {
   public double d1, d2, d3;
   public scene_prop6 (double d1,double d2,double d3) {this.d1=d1;this.d2=d2;this.d3=d3;}
   public override void translate (StreamWriter asmFile) {
      asmFile.WriteLine("Push\t{0}", d1);
      asmFile.WriteLine("Push\t{0}", d2);
      asmFile.WriteLine("Push\t{0}", d3);
      asmFile.WriteLine("Ambient");
      asmFile.WriteLine("Axes\n");
   }
}
...%node scene_prop6 : scene_prop {
   public double d1, d2, d3;
   public scene_prop6 (double d1,double d2,double d3) {this.d1=d1;this.d2=d2;this.d3=d3;}
   public override void translate (StreamWriter asmFile) {
      asmFile.WriteLine("Push\t{0}", d1);
      asmFile.WriteLine("Push\t{0}", d2);
      asmFile.WriteLine("Push\t{0}", d3);
      asmFile.WriteLine("Ambient");
      asmFile.WriteLine("Axes\n");
   }
}

...%node scene_prop6 : scene_prop {
   public double d1, d2, d3;
   public scene_prop6 (double d1,double d2,double d3) {this.d1=d1;this.d2=d2;this.d3=d3;}
   public override void translate (StreamWriter asmFile) {
      asmFile.WriteLine("Push\t{0}", d1);
      asmFile.WriteLine("Push\t{0}", d2);
      asmFile.WriteLine("Push\t{0}", d3);
      asmFile.WriteLine("Ambient");
      asmFile.WriteLine("Axes\n");
   }
}

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
   %scene_prop6 (d1.val,d2.val,d3.val);

prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp ...
```
Adding the ambient Construct to a Hard-coded DSL VM

- Adding the Ambient instruction to the VM’s lexer definition

```plaintext
%token DOUBLE {public double val;}
...
"Window"                   %WIN
"Light"                    %LIGHT
"COP"                      %COP
"LookAt"                   %LOOK
"VUP"                      %VUP
"Ambient"                  %AMB
...
```
Adding the ambient Construct to a Hard-coded DSL VM

- Parsing the new Ambient instruction

```java
...  
@node asm9_amb : asm
{
    public override void interpret (RTVM.RTMachine rtm)
    {
        rtm.RT_AMB ();
    }
}
...

asm_list : %asm_list1 (null, null);
asm_list : asm_list:al asm:a %asm_list1 (al, a);

asm : PUSH DOUBLE:d %asm1_push (d.val);
asm : SPHERE %asm2_sphere ();
asm : PLANE %asm3_plane ();
asm : WIN %asm4_win ();
asm : LIGHT %asm5_light ();
asm : COP %asm6_cop ();
asm : LOOK %asm7_look ();
asm : VUP %asm8_vup ();
asm : AMB %asm9_amb ();
asm : GEOM %asm10_geom ();
...```
Adding the ambient Construct to a Hard-coded DSL VM

- The semantics for the new Ambient instruction in function RT_AMB():

```java
class RTMachine {
    public RTMachine (int rd, int stackSize) {
        rtstack = new RTStack (stackSize);
        scene = new Scene ();
        render_depth = rd;
    }
    ...
    // Alter the current ambient colour vector.
    public void RT_AMB () {
        double amb = (double) rtstack.pop ();
        double amg = (double) rtstack.pop ();
        double amr = (double) rtstack.pop ();
        scene.alterAmbient (amr, amg, amb);
    }
    ...
    private asm_list ast;
    public RTStack rtstack;
    public Scene scene;
    public int render_depth;
}
```
 DSLs using the VM Framework

- Conceptual view of a VM using the VM Framework

```
scene
{
  sphere 0.5
  {
    translate 3 3 0
    specular 0 0 1
  }
}
```

```
{ sphere 0.5
  {
    translate 3 3 0
    specular 0 0 1
  }
}
```
The VM Framework

- Designed on the .NET environment, written in C#

- The VM Framework provides a Virtual Machine with an empty instruction set

- The VM is extended by a set of concrete instructions and environments to facilitate the semantics of a domain
The VM Framework

EVM
- temps:Hashtable
- Reset()
- Execute():object
  - retrieves result
  - executes
  - loads program
  - configures

Env
- GetResult():object

Config
- instructions:Hashtable
- environments:Hashtable
- env_ins:Hashtable
  - has available
  - provides environment and instructions

Type
  - operates on
  - has available

Loader
- Program_counter:int
- Reset()
  - loads
  - configures

Inst
- Execute()
Once instantiated, it represents a VM, that can be configured and execute a loaded IS program.

EVM
- temps:Hashtable
- Reset()
- Execute()
- GetResult():object

Env
- GetResult():object

Config
- instructions:Hashtable
- environments:Hashtable
- env_ins:Hashtable

Type
- 1, 1..*

 Loader
- Program_counter:int
- Reset()

Inst
- 1, 1..*
- Execute()
The VM Framework

- **EVM**
  - temps:Hashtable
  - Reset()
  - Execute():object
- **Env**
  - GetResult():object
- **Config**
  - instructions:Hashtable
  - environments:Hashtable
  - env_ins:Hashtable
  - Reads a configuration file, and maps environment names to an instantiated environment, and maps instructions to instruction classes.
- **Type**
  - has available
- **Loader**
  - Program_counter:int
  - Reset()
- **Inst**
  - Execute()
The VM Framework

EVM
- temps:Hashtable
- Reset()
- Execute():object

Env
- GetResult():object

Config
- instructions:Hashtable
- environments:Hashtable
- env_ins:Hashtable

Type
- has available
- type of

Loader
- Program_counter:int
- Reset()

Inst
- Execute()
The VM Framework

EVM
- temps:Hashtable
- Reset()
- Execute():object
  - retrieves result
  - executes

Env
- GetResult():object
  - 1 1..*

Config
- instructions:Hashtable
- environments:Hashtable
- env_ins:Hashtable
  - has available
  - provides environment and type of instructions

Loader
- Program_counter:int
- Reset()
  - loads
  - program

Type
- 1 1..*

Inst
- Execute():object
  - 1 1..*

Extensions of Env, encapsulates some ADT appropriate for the DSL.
The VM Framework

- **EVM**
  - temps: Hashtable
  - Reset()
  - Execute(): object

- **Env**
  - GetResult(): object

- **Config**
  - instructions: Hashtable
  - environments: Hashtable
  - env_ins: Hashtable

- **Type**
  - has available

- **Loader**
  - Program_counter: int
  - Reset()

- **Inst**
  - Execute()

- **Extensions of Inst update an Env instance. Instantiated according to the operand of the instruction in the source IS file.**
The VM Framework

- Environments
  - An ADT updated during runtime.
  - Intended to aid in the processing of the semantics of the DSL in an appropriate or convenient manner. For example, in a simple real-valued expression language can use a runtime stack.
  - Classes that extend the abstract `Env` class, are obligated to override the method `GetResult()`.
Example `EnvExp` class

class EnvExp : Env
{
    public EnvExp () {
        stack = new Stack (100);
    }
    public override object GetResult () {
        object result;
        if (stack.isEmpty ()) {
            result = -1.0;
        } else {
            result = stack.peek ();
        }
        return result;
    }
    public Stack GetStack () { return stack; }
    protected Stack stack;
}
Instructions
- There are 5 classes of instructions, defined in terms of their operand types.
- The instructions form the basic operational building blocks to a DSL-compiled program.
- Each instruction is allowed to take at most one operand.
- Classes that extend one of the 5 instruction classes, are obligated to override the method `Execute()`.
The Five Classes of Instructions

- **Inst**
  - Execute()

- **Inst_OpCode**
  - LoadProgramCounter:bool

- **Inst_OpCode_Br**
  - label:string
  - BranchCond:bool
  - SaveProgramCounter:bool

- **Inst_OpCode_ID**
  - ID:string
  - temp:object

- **Inst_OpCode_Num**
  - num:double

- **Inst_OpCode_Str**
  - str:string
The Five Classes of Instructions

- **Inst**
  - Execute()
  
- **Inst_OpCode**
  - **Inst_OpCode_Br**
    - label: string
    - BranchCond: bool
    - SaveProgramCounter: bool
  
  - **Inst_OpCode_ID**
    - ID: string
    - temp: object
  
- **Inst_OpCode_Num**
  - num: double

**Inst_OpCode_Str**
- str: string

Instantiated when an instruction has no operand.
The Five Classes of Instructions

- Inst
  - Execute()

- Inst_OpCode
  - LoadProgramCounter:bool

- Inst_OpCode_Br
  - label:string
  - BranchCond:bool
  - SaveProgramCounter:bool

- Inst_OpCode_ID
  - ID:string
  - temp:object

- Inst_OpCode_Num
  - num:double

- Inst_OpCode_Str
  - str:string

Instantiated when the instruction has a branch label following it.
The Five Classes of Instructions

Instantiated when the instruction has a reference to an internal temporary value.
The Five Classes of Instructions

- **Inst**
  - Execute()

- **Inst_OpCode**
  - LoadProgramCounter: bool

- **Inst_OpCode_Br**
  - label: string
  - BranchCond: bool
  - SaveProgramCounter: bool

- **Inst_OpCode_ID**
  - ID: string
  - temp: object

- **Inst_OpCode_Num**
  - num: double

- **Inst_OpCode_Str**
  - str: string

Instantiated when the instruction has a number (real or integer) following it.
The Five Classes of Instructions

- Inst
  - Execute()

- Inst_OpCode
  - LoadProgramCounter:bool

- Inst_OpCode_Br
  - label:string
  - BranchCond:bool
  - SaveProgramCounter:bool

- Inst_OpCode_ID
  - ID:string
  - temp:object

- Inst_OpCode_Num
  - num:double

- Inst_OpCode_Str
  - str:string
  - Instantiated when the instruction has a string following it.
The Configuration File

- The config file defines an ISA for a particular VM instance, in essence defining an instruction pool, each instruction to be later instantiated by loading a DSL-compiled program.
- For each instruction specified in the config file, there needs to be a compiled DLL module on disk with the same name, which contains the compiled instruction class.
Extending the VM Framework

• Example configuration file for EXP

```plaintext
(* Create an instance of the *)
(* expression environment. *)
environment EnvExp

(* Register the following expression
(* instructions with the DVM. *)
Push  using EnvExp
Store using EnvExp
Load  using EnvExp
Sub   using EnvExp
Add   using EnvExp
Mul   using EnvExp
Br    using EnvExp
Brgz  using EnvExp
Nop   using EnvExp
Div   using EnvExp

(* Some generic instructions *)
Call  using EnvExp
Ret   using EnvExp
Print using EnvExp
```
Extending the Environments

- Environment EnvRT as a subtype of EnvExp

```csharp
class EnvRT : EnvExp
{
    public EnvRT () : base ()
    {
        scene = new Scene ();
    }

    public override object GetResult ()
    {
        scene.render (4);
        return scene.GetBitmap ();
    }

    protected Scene scene;
    protected Bitmap bitmap;
}
```
Extending the Instruction Sets

- Defining an environment and instruction set for a DSL VM

---

**Diagram:**

- **ISA_{EXP}**
- **Dynamicly Interpret IS**
- **VM_{EXP}**
- **Execute on CLR**
- **HLL VM**

**Configuration:**

- environment EnvExp
- Push using EnvExp
- Add using EnvExp
- Mul using EnvExp
- Ret using EnvExp
- ...
ISA_{LQ} is a subset of ISA_{EXP}, with new instructions.

**Configuration**
- `VM_{EXP}`
- `VM_{LQ}`
- `Dynamically Interpret IS`
- `Execute on CLR`
- `HLL VM`
- `environment EnvExp`
- `QuiltA using EnvExp`
- `QuiltB using EnvExp`
- `Turn using EnvExp`
- `Ret using EnvExp`
- `...`
$ISA_{RT}$ is a subset of $ISA_{EXP}$, with new instructions as before.
Adding the ambient Construct using the VM Framework

- Altering the grammar for the new ambient construct on the RT compiler

```java
%node scene_prop6 : scene_prop
{
    public double d1, d2, d3;
    public scene_prop6 (double d1, double d2, double d3)
    {
       this.d1 = d1; this.d2 = d2; this.d3 = d3;
    }
    public override void translate (StreamWriter asmFile)
    {
       asmFile.WriteLine("Push\{0\}", d1);
       asmFile.WriteLine("Push\{0\}", d2);
       asmFile.WriteLine("Push\{0\}", d3);
       asmFile.WriteLine("Ambient");
       asmFile.WriteLine("Axes\n");
    }
}
...

element : scene_prop
element : prim:p LBRACE prim_prop_list:ppl RBRACE

scene_prop : WINDOW DOUBLE:d1 DOUBLE:d2
scene_prop : LIGHT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : COP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : LOOKAT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : VIEWUP DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
scene_prop : AMBIENT DOUBLE:d1 DOUBLE:d2 DOUBLE:d3
    %scene_prop6 (d1.val, d2.val, d3.val);
prim_prop_list :
prim_prop_list : prim_prop_list:ppl prim_prop:pp
...
Adding the ambient Construct using the VM Framework

- The Ambient instruction extends Inst_OpCode (no operands)

```csharp
public class Ambient : Inst_OpCode
{
    public Ambient (EnvRT env)
    {
        this.env = env;
    }

    public override void Execute ()
    {
        double amz = (double) ((EnvRT) env).stack.pop();
        double amy = (double) ((EnvRT) env).stack.pop();
        double amx = (double) ((EnvRT) env).stack.pop();
        ((EnvRT) env).scene.alterAmbient (amx, amy, amz);
    }
}
```
Adding the ambient Construct using the VM Framework

- Adding the Ambient instruction to the instruction set

```haskell
(* Create an instance of the ray-tracer environment. *)
environment EnvRT

...
COP       using EnvRT
LookAt    using EnvRT
VUP       using EnvRT
Ambient   using EnvRT
Geometry  using EnvRT
Specular  using EnvRT
Diffuse   using EnvRT
...
```
DSL Examples
The Domain of EXP

+  -  *  ÷
The Environment of EXP

**Env**
- GetResult():object

**EnvExp**
- GetResult():object

**Stack**
- stack:object[]
- push()
- pop():object
- peek():object
- isEmpty:bool

---

**run-time stack**
The ‘Add’ Instruction

```csharp
public override void Execute()
{
    double d1;
    double d2;
    d2 = (double)((EnvExp) env).stack.pop();
    d1 = (double)((EnvExp) env).stack.pop();
    ((EnvExp) env).stack.push(d1 + d2);
}
```
EXP Example

Configuration file:

```plaintext
environment EnvExp

Push  using EnvExp
Store using EnvExp
Load  using EnvExp
Sub   using EnvExp
Add   using EnvExp
Mul   using EnvExp
Br    using EnvExp
Brgz  using EnvExp
Nop   using EnvExp
Div   using EnvExp

Call  using EnvExp
Ret   using EnvExp
Print using EnvExp
```
let
  n = 5
in
  9 + sum (i) 1..n (2 * i)
end

EXP Example
let
  n = 5
in
  9 + sum (i) 1..n (2 * i)
end

Push 5
Store $1
Push 9
Push 1
Store $2
Push 0
@100 Load $2
Load $1
Sub
  Brgz label @200
Push 2
Load $2
Mul
Add
Load $2
Push 1
Add
Store $2
Br label @100
@200 Nop
Add
EXP Example

Output result: System.Double
The Domain of RT

\[ q(x, y, z) = a_{11}x^2 + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + 2a_{12}xy + b_1x + b_2y + b_3z + c = 0 \]
The Environment of RT

Env
GetResult():object

EnvExp
GetResult():object

EnvRT
GetResult():object

Stack
stack:object[]
push()
pop():object
peek():object
isEmpty:bool

Scene
render()
ShootRay()

Bitmap

objects in scene

run-time stack

result canvas
RT Example

Configuration file:

```plaintext
environment EnvRT

Push using EnvRT
Add using EnvRT
Sub using EnvRT
Mul using EnvRT
Div using EnvRT

SetWindow using EnvRT
Light using EnvRT
COP using EnvRT
LookAt using EnvRT
VUP using EnvRT
Ambient using EnvRT
Geometry using EnvRT
Specular using EnvRT
Diffuse using EnvRT
Reflect using EnvRT
Refract using EnvRT
Translate using EnvRT
Axes using EnvRT
Coeff using EnvRT
Quad using EnvRT
```
scene {
  window 300.0 200.0  light -3.5 0.5 2.0
  cop 0.0 4.0 1.0  lookat 0.0 0.0 0.0
  viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

  sphere 0.5 {
    translate 1.0 0.0 0.0  reflective 0.1 0.1 0.1
    specular 0.7 0.0 0.0  diffuse 0.7 0.0 0.0
  }

  sphere 0.5 {
    translate 0.0 -1.0 0.2  reflective 0.2 0.3 0.2
    specular 0.0 0.7 0.0  diffuse 0.0 0.7 0.0
  }

  sphere 0.5 {
    translate -1.0 -0.5 0.0  reflective 0.1 0.1 0.1
    specular 0.0 0.0 0.7  diffuse 0.0 0.0 0.7
  }

  plane -0.1 0.0 1.0 1.0 {
    specular 0.0 0.0 0.0  diffuse 1.0 0.5 0.0
    reflective 0.25 0.25 0.25
  }
}

Coeff "c"

Push 1
Push 0
Push 0
Translate
RT Example

```plaintext
scene {
    window 300.0 200.0  light -3.5 0.5 2.0
    cop 0.0 4.0 1.0  lookat 0.0 0.0 0.0
    viewup 0.0 0.0 1.0  ambient 0.3 0.3 0.3

    sphere 0.5 {
        translate 1.0 0.0 0.0  reflective 0.1 0.1 0.1
        specular
    }Quad
    Push 1
    Push 0.5
    Push 0.5
    Mul
    Div
    Coeff "a11"
    Push 1
    Push 0.5
    Push 0.5
    Mul
    Div
    Coeff "a22"
    Push 1
    Push 0.5
    Push 0.5
    Mul
    Div
    Coeff "a33"
    Push -1
    Coeff "c"
    Push 1
    Push 0
    Push 0
    Translate
}
```
RT Example

Output result: System.Drawing.Bitmap
The Domain of LQ

LittleQuilt

enum Orientation
{
  N,
  E,
  S,
  W
}

denum Symbol
{
  A,
  B
}

Quilt
NumRows: uint
NumCols: uint
quilt: Orientation [,]
symbols: Symbol [,]
Right(): Quilt
Sew(): Quilt

A = 
B =
The Environment of LQ

Env

EnvExp

Stack

GetResult():object

push()
pop():object
peek():object
isEmpty:bool
LQ Example

Configuration file:

```
environment EnvExp
QuiltA using EnvExp
QuiltB using EnvExp
Load using EnvExp
Store using EnvExp
Turn using EnvExp
Sew using EnvExp
Call using EnvExp
Ret using EnvExp
Br using EnvExp
```
let
  fun unturn (x) = turn (turn (turn (x)))
  fun pile (x, y) = unturn (sew (turn (y), turn (x)))
  val s = turn (sew (unturn (unturn (a)), b))
  val t = pile (turn (b), unturn (b))
  val u = turn (turn (sew (s, t)))
  val m = sew (turn (a), turn (turn (a)))
  val n = sew (unturn (a), a)
  val p = pile (m, n)
  val q = sew (sew (u, p), turn (u))
  in
  pile (q, turn (turn (q)))
end
let

fun unturn (x) = turn (turn (turn (x)))

fun pile (x, y) = unturn (sew (turn (y), turn (x)))

val s = turn (sew (unturn (unturn (a)), b))

val t = pile (turn (b), unturn (b))

val u = turn (turn (sew (s, t)))

val m = sew (turn (a), turn (turn (a)))

val n = sew (unturn (a), a)

val p = pile (m, n)

val q = sew (sew (u, p), turn (u))

in

pile (q, turn (turn (q)));

end
LQ Example

Output result: LittleQuilt.Quilt

```plaintext
let
  fun unturn (x) = turn (turn (turn (x)))
  fun pile (x, y) = unturn (sew (turn (y), turn (x)))
  val s = turn (sew (unturn (unturn (a)), b))
  val t = pile (turn (b), unturn (b))
  val u = turn (turn (sew (s, t)))
  val m = sew (turn (a), turn (turn (a)))
  val n = sew (unturn (a), a)
  val p = pile (m, n)
  val q = sew (sew (u, p), turn (u))
in
  pile (q, turn (turn (q)))
end
```
Comparative Results

Legend

1 - Interpreter
   - SRC -> SEM (Interpreter)
2 - Hard-coded VM
   - SRC -> IR
3 - VM Framework
   - IR -> SEM
<table>
<thead>
<tr>
<th></th>
<th>Scalability</th>
<th>Performance</th>
<th>Modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interpreter</strong></td>
<td>◼️: Difficult and cumbersome to modify a language and alter semantics.</td>
<td>◼️: Interpretation of AST done in same program, and there’s no IS.</td>
<td>◼️: Syntax and semantics issues are fairly dependant on one another.</td>
</tr>
<tr>
<td><strong>Hard-coded VM</strong></td>
<td>◼️: Instructions are still required to be hard-coded as part of the core.</td>
<td>◼️: Compiled program needs to be parsed before it can be executed.</td>
<td>◼️: Minimal effort required adding new instructions to the VM core.</td>
</tr>
<tr>
<td><strong>VM Framework</strong></td>
<td>◼️: Easily add new instructions and environments as the language grows.</td>
<td>◼️: Comparable performance to the hard-coded VM, but slightly faster for compiled program execution.</td>
<td>◼️: Instructions and environments form the link between the VM and the underlying namespace, and are easily modified as the namespace expands.</td>
</tr>
</tbody>
</table>
Thank You