Runtime Code Generation in Strongly Typed Execution Environments

Antonio Cisternino

28/3/2003

Supported by Microsoft Research grant

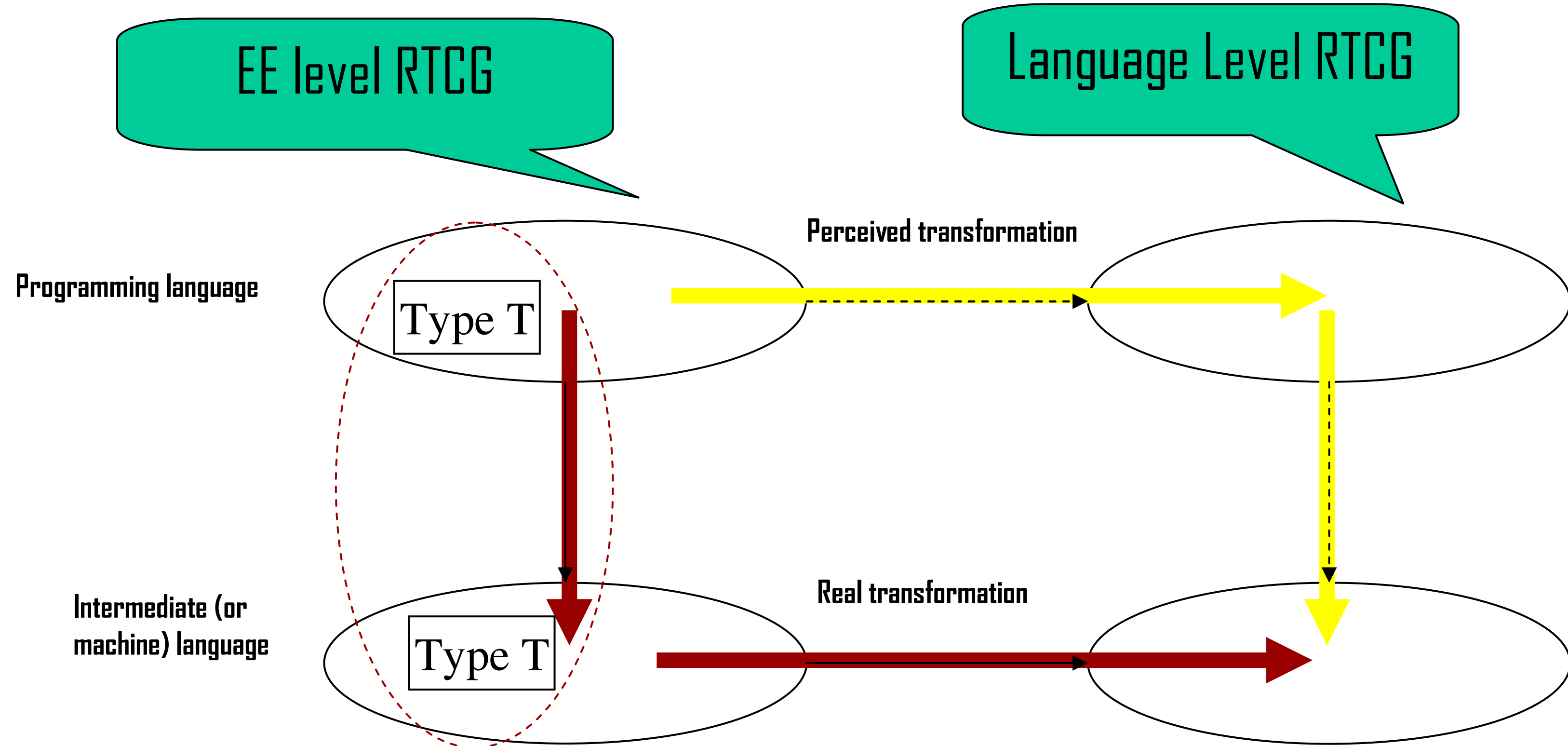
CODE BRICKS

Introduction

- Dynamic code generation is a programming technique that is being widely used in complex systems
- Strongly Typed Execution Environments (STEE), such as JVM and Microsoft CLR, provide rich services to running code (dynamic loading, security, …)
- To enforce type safety at load time STEE should retain enough information in binaries about types (metadata)
- It is convenient for PL targeting STEE to share a significant amount of programming abstractions

CODE BRICKS

Encoding Code Generation into Binaries



What we did?

- We introduced a **Code** type to represents code values
- Code instances are built around methods
- A single operation called **Bind** allow code manipulation

```
public static int add(int i, int j) {
  return i + j; }
// λx,y.x+y
Code c = new Code(typeof(T).GetMethod("add"));
// λy.1+y
Code inc = c.Bind(1, new Free());
```

Binding Values (Bind_v)

- Values can be bound to arguments
- All values can be bound to an input argument:

```
Code c = SelectCodeGen();
DBConnection conn = new Connection(…);
c.Bind(conn);// Fix the connection
```

Binding Code Values to Args (Bind$_\pi$)

- An input argument of a code value can be bound with another code value returning a value (compatible with it)
- Arguments of the code value bound to an argument are lifted into the signature of the resulting code value

```
Code c = CodeOfAdd(); // c is λx,y.x+y
Code d = c.Bind(c, new Free());
// d is λx,y,z.x + y + z
```

Splicing Code Values (Bind$_\chi$)

- If the argument is of higher order type (a delegate type in CLI) we can splice a code value in place of invocations of the argument
- With this mechanism it is possible to write methods with holes for other methods

```
delegate void Cmd(int i);
static void For(Cmd c) {
  for (int j; j < 10; j++) c(j); }
static void Print(int i) {
  Console.WriteLine("Iteration {0}", i); }
Code f = CodeOfFor(), p = CodeOfPrint();
Code c = f.Bind(new Splice(p));
```

Formal Model

- Let
  - c = addCode.Bind(addCode.Bind(2, new Free()), new Free()),

we expect:

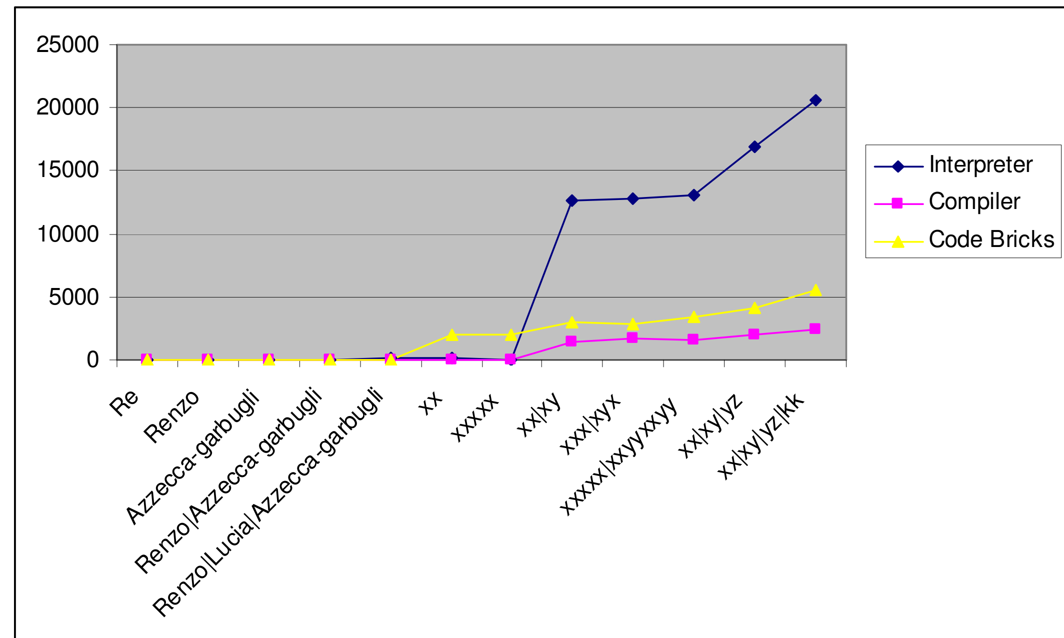
  - add(add(2, x), y) == c(x, y)
- We have defined a transformation on IL code that "cut and paste" code fragments
- Theorem: the code generated by the transformation preserves the expected semantics
- Theorem: only well-formed and type-safe code can be generated

Applications

- Bind$_{\pi}$ is useful to turn interpreter in dummy compilers
- Our current implementation is pretty fast: it is suitable for runtime code generation
- General combinators can be defined to generate arbitrary programs
- It is possible to express staged computations because transformation is IL -> IL

RE: Benchmark

- It is a *worst case* test: we use debug code!!!
- Compilation time of Code Bricks version is about one third of the SSCLI version



Conclusions

- Code generation is expressed by means of combination of code values (basically methods)
- Method application notion offers control over the generated code
- It is possible to express staged computations that can even be 'cross process'
- It is possible to derive properties of generated code relying on the structure of the binding