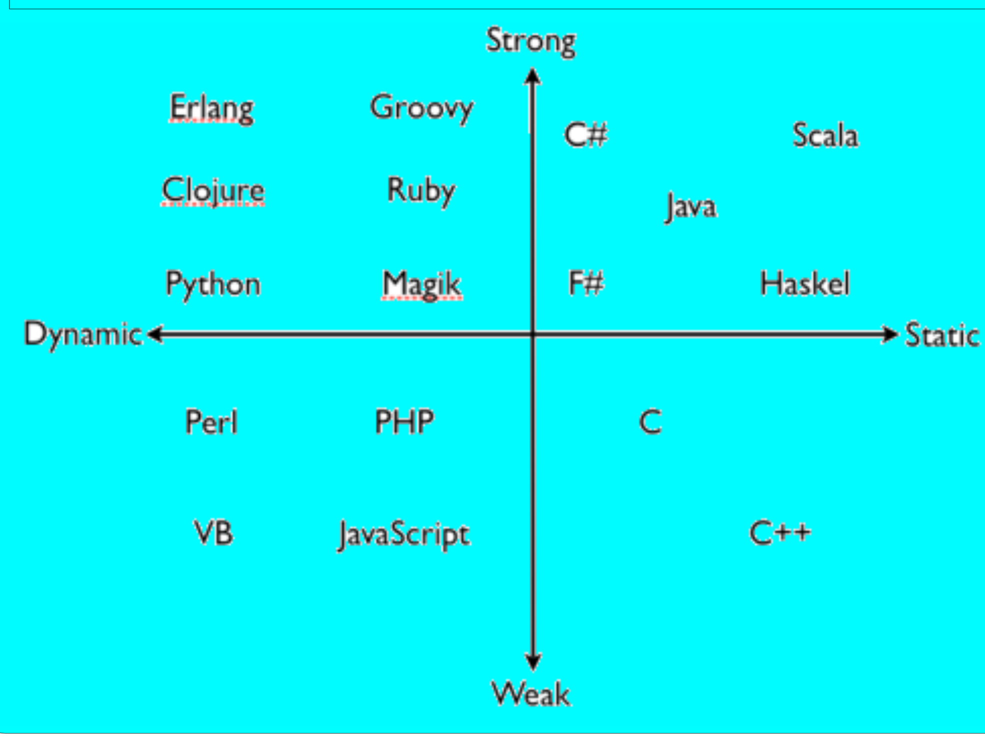
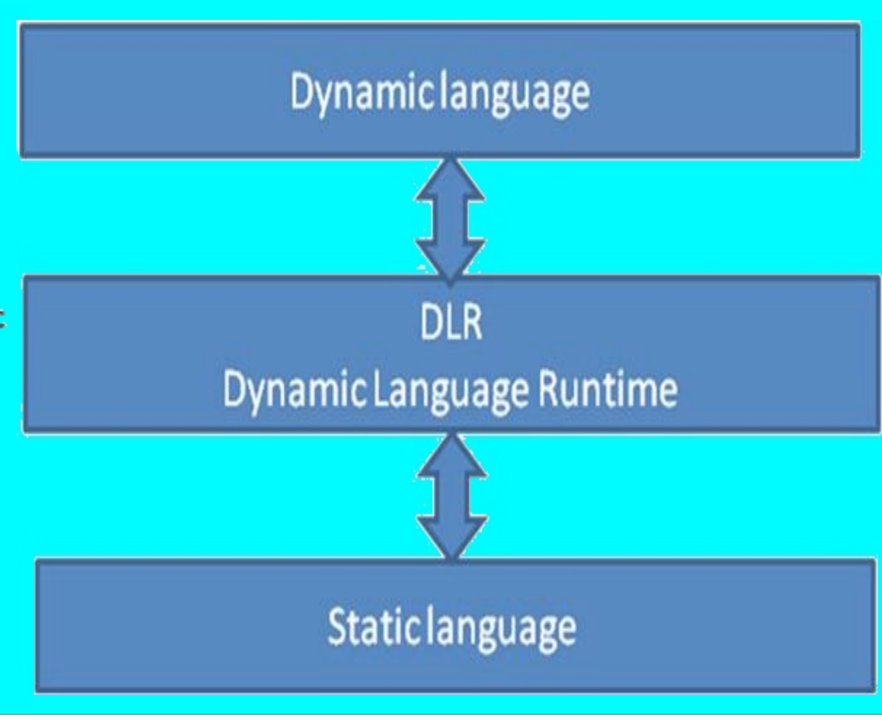
Waterfall JIT Compiler

Guido Chari – gchari@dc.uba.ar
Departamento de Comptación, FCEyN, Universidad de Buenos Aires

Dynamic Programming Languages





Properties that Affect Execution Time

- ✓: Memory management (Garbage Collection).
- ✓: Polymorphism (Late binding, dispatching).
- ✓: Object Creation (Object Format).
- ✓: Code interpretation and/or generation.

Some Context and Comparison

SLANG

Waterfall translates from Slang language to native code. Slang can be described with some few statements:

- ✓ **S1**: The syntaxis is a subset of ST.
- ✓ **S2:** Only basic types.
- ✓ **S3:** No polymorphism.
- ✓ **S4:** No runtime (GC, Dispatch, etc.).
- ✓ **S5:** It is translated to C and compiled with standard C compilers.

Pharo and Squeak Virtual Machines are written with this language. The code is live code inside **Smalltalk** standard images. For compilation, the code scattered in lot of Smalltalk methods is **translated to C.** Finally, after compiling with standard **C compilers** a binary is generated. Slang is **efficient** since it is **Static**.

Dynamic Languages	Static Languages
Simple and succinct	Robust
Implicitly typed	Performant
Meta-programming	Intelligent tools
No compilation	Better scaling

Hypothesis

In practice, what can be found in general, is that when software models become mature enough these properties holds:

- ✓ **S1**: Monomorphic dispatchs.
- ✓ **S2:** Reduced type explosion.
- ✓ **S3:** Feasible to infer types.

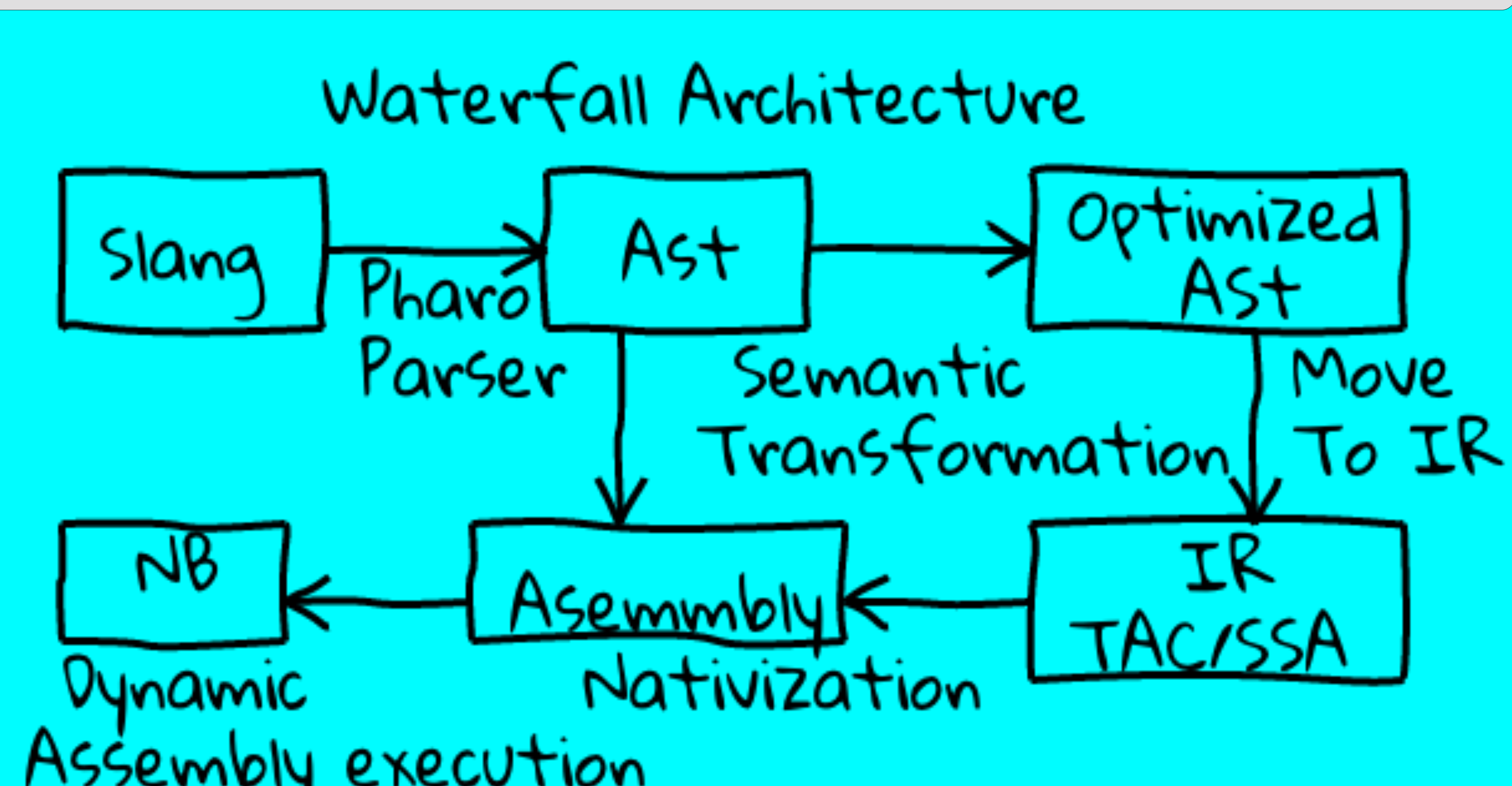
So… what can we do for **exploiting this properties** and for having software that avoids dynamic execution overheads?

With a **dynamic Just In Time Compiler** that could **transparently** try to maximize the amount of code that could be statically compiled, some particular code sections could be **late transformed** (on demand and inside the language) but run efficiently by avoiding the VM runtime overhead. The solution would also avoid resorting to two-language approaches.

Development and Architecture

In order to experiment with our hypothesis we decided to develop a dynamic JIT for translating SLANG code into native one. Also we developed a transparent API and extend existing frameworks for dynamic binary code execution inside a runtime. This are the most prominent stages of the compiler:

- Generate an Abstract Syntax Tree of the method to be compiled.
- Transform the AST to avoid incompatibilities and restrict a little the spectrum of the compilable SLANG.
- Translate the AST to a general purpose IR suitable for optimizations. This IR is also executable by an interpreter.
- Nativize the method by traversing adequately the IR representation. This stage must be repeated for any method that is referenced. Also all SLANG primitive operations references must be compiled. This stage could also be done at the AST avoiding the IR step.
- Use NativeBoost (a general dynamic high-level low-level programming framework) for the generation and execution of the assembly.



Done and Next

What's been done:
After the development of the JIT infrastructure we started with some experiments. Mainly we focused on the idea of using Waterfall JIT for improving or evolving the VM. Instead of changing the VM low-level source code our intention was to do it in the same language that it supported. That means ***Reflection*** at VM level instead of at language-side. For validating the idea we decided to try changing VM primitives at runtime. To make the experiment more real, we frame it into an instrumentation experiment. It is well known that instrumenting primitives of the VM at language side is so inefficient that it sometimes is even prohibitive.
Actually, we are finishing to move some ***plugins*** from VM to language side. Linked to binaries, by moving them to language side, they could be **dynamically compiled** at demand (lazily) favoring dynamic auto customizable VMs with cleaner, smaller and more cohesive kernels.
Here some benchmarks of the primitive instrumentation experiments done that are really encouraging:

Tool	Running Time	Relative Difference
VM	6.4 +/ 0.14ms	1.0x
Waterfall	22.8 +/- 0.17ms	~3.6x
Reflection	195 +/- 0.16ms	~30x

What's next:

- More experiments.
- Type Inference
- More IRs.
- Code analysis and optimizations.
- Inteface for interaction with standard compilation frameworks (LLVM).
- More IRs.
- Extend the possibilities of usage.
- Reflective runtimes.