Optimization axis for interpreter

Benoit Zanotti

jicks@lse.epita.fr
http://www.lse.epita.fr

July 17, 2012

Plan

Plan

Definition

Definition

An interpreter is a computer program that executes instruction written in a programming language.

Wide definition → different way of interpreting.
The source code can be translated/compiled before being executed.

Interpreting categories

- Direct execution of the source code (*Lisp*).
- Translation to an intermediate representation, which is executed (*Perl*, *Ruby*).
- Precompilation of the source code and execution.

It is possible to combine those categories (*Java*).

Plan

Basic mechanism of the language

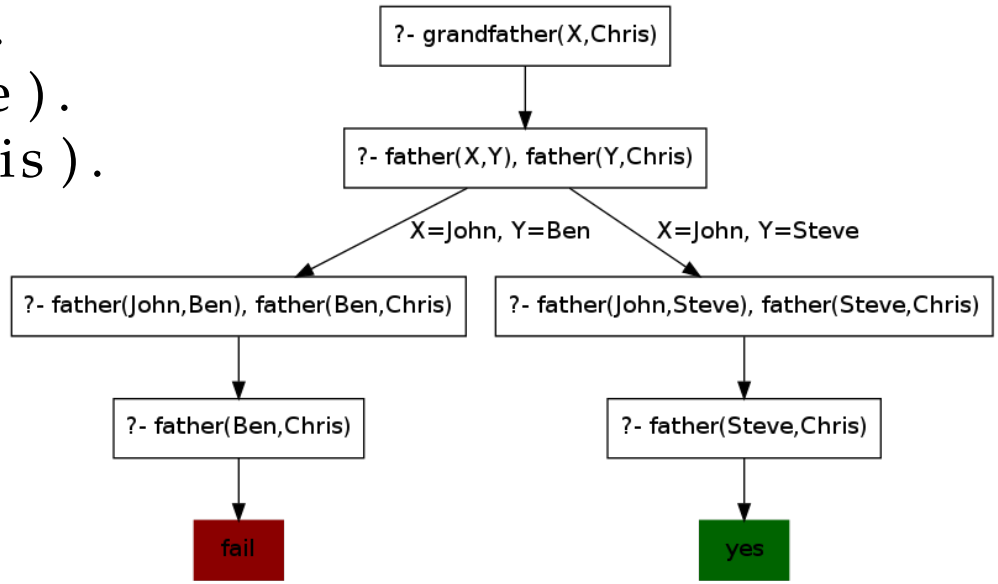
- Logic programming (**Pro**grammation **Log**ique) paradigm.
- Based on clauses (rules and facts) and queries.
  - `student(zanott_b).`
  - `father(X) :- parent(X,Y), male(X).`
  - `?- father(zanott_b)`
- Variables in queries to get informations.
  - List of every known father:

    `?- father(X)`

Search tree

- A Prolog query resolution can be seen as a search tree.
- Several branches = several clauses "version".

```
grandfather(X,Y) :- father(X,Z), father(Z,Y).
father(John,Ben).
father(John,Steve).
father(Steve,Chris).
```



Mechanism of the interpreter

A Prolog interpreter is composed of two major parts:

- The **knowledge base**, storing facts, rules and associations.
- The **inference engine**, using the knowledge base to derive answer to users query.
- To find to which "version" of a clause we have to expand, we try to "unify" them.

Why is Prolog not a good example

Exotic paradigm and system of execution (everything uses the inference engine):

- Optimizations will differ from more conventional language.
- Some usual optimizations can't be used.

But

- The axis of optimizations are still the same.

Plan

Plan

Advantages

Interpreted languages holds some advantages against compiled languages:

- Easier development cycles.
- Platform independance.
- Reflection and introspection.
- Dynamic typing and scoping.

Inconvenients

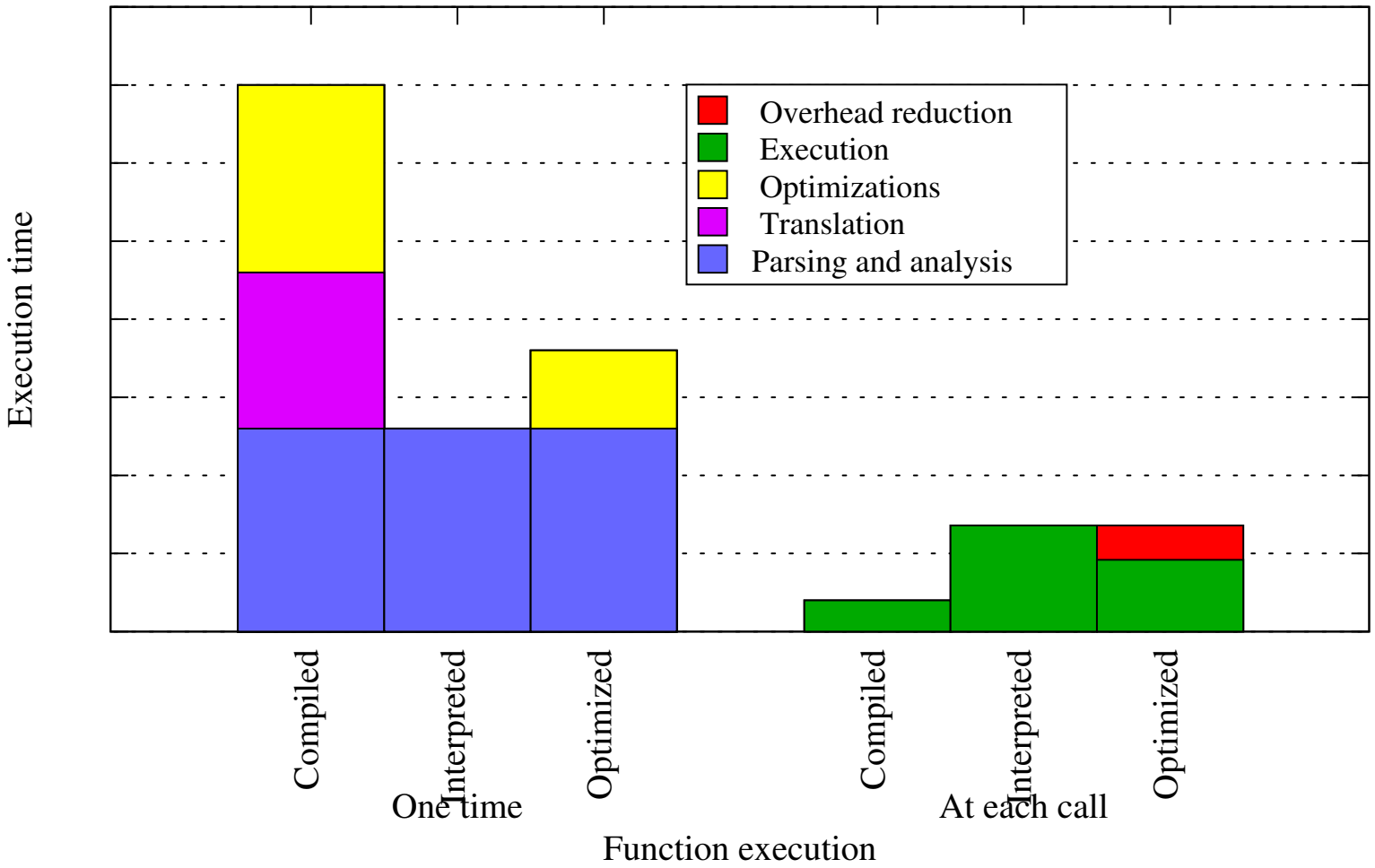
There is only one major inconvenient: the **speed**

- Overhead because each statement must be analyzed before execution.

Main problem: reducing this overhead as much as possible.

Plan

Analyzing the overhead



Limitations of the optimizations

Because optimizations are done at run-time, we have a dilemma:

- We can reduce execution time by doing optimization.
- We can reduce optimization time by doing less of them.

→ We want to balance those two aspects.

Balancing is hard

- To achieve the perfect equilibrium, we need to know the input (either impossible or useless).
- We use **heuristics** to know when we must optimize or not.
- Searching for **hotspots** (inner-most loops, ...) : most of the time is spent executing them.

Plan

Plan

Static optimization

- Static optimization refers to all that can be done without any runtime informations.
- Compilers use them extensively and the theory behind them is well researched.
- They cover many areas (mainly machine-independant for interpreter): optimizing the common case, reducing redundency, doing some strength reduction, ...

Advantages and limitations

- ✓ Some of the optimizations can be **really** effective.
- ✗ Some dynamic language are impervious to most optimizations.
- ✗ Runtime informations allow the use of more specific and precise optimizations.

Plan

Example in Prolog

- Variables tend to make the unification algorithm succeed.
- The sooner the unification fails, the better (all subbranches won't be visited).
- Many clauses can be reordered with the same meaning.

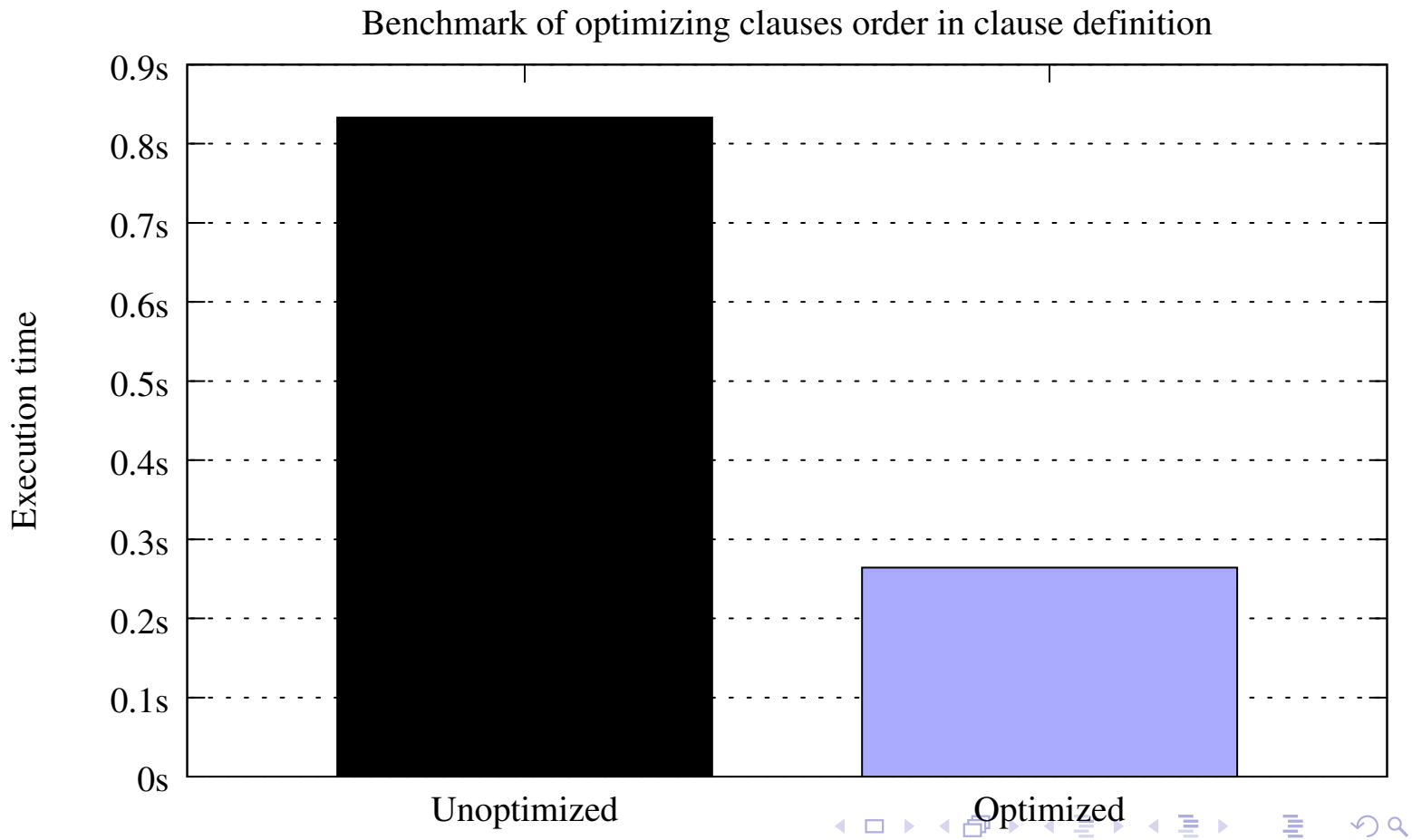
→ We use a *heuristic* to reorder clauses to assign variables as early as possible.

Example in Prolog

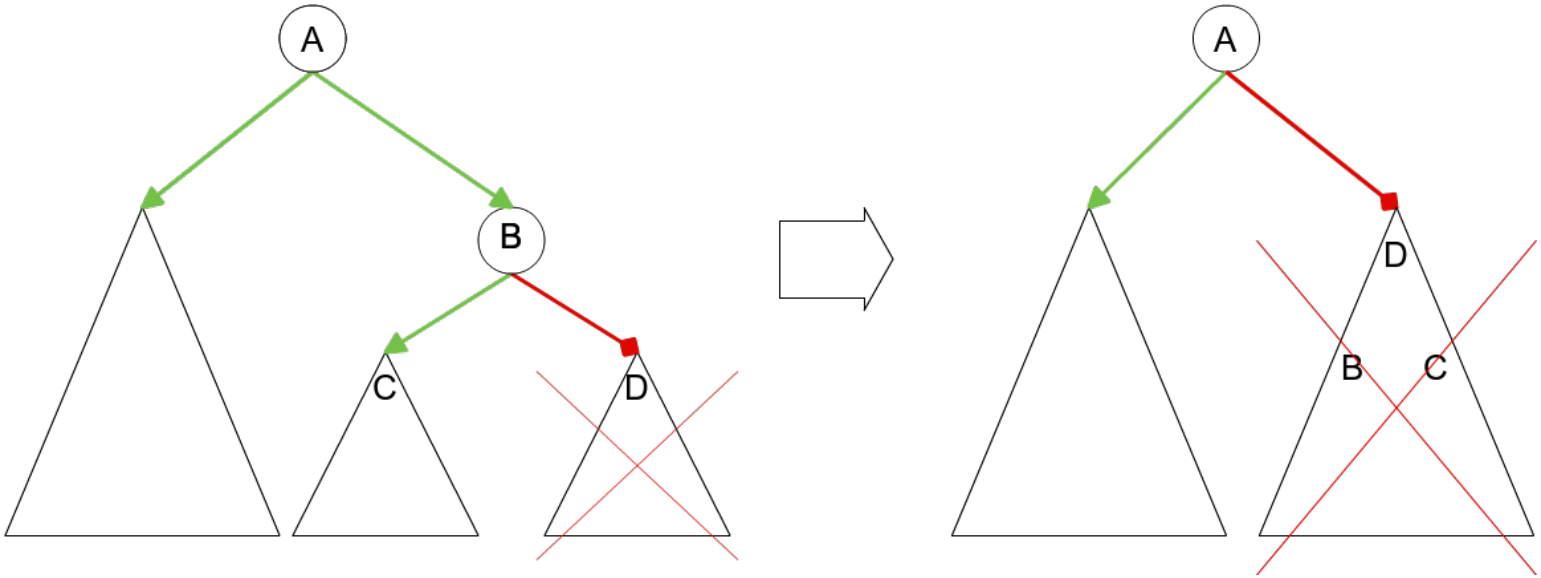
```
foo(X) :- bar1(X),bar2(Z),bar4(Y,X),baz(X,Y,Z).
```

becomes

```
foo(X) :- baz(X,Y,Z),bar4(Y,X),bar1(X),bar2(Z).
```



Explanation of the example



- Unification failure → subbranch **not covered**.
- We want those failure as hight as possible in the tree.

Example in compilers

- Loop optimizations (unrolling, loop-invariant code motion, ...).
- Data-flow optimizations (constant folding, elimination of common subexpression, ...).
- SSA (Static Single Assignment) based optimizations.
- Inlining, test reordering, ...

Plan

4. Run-time Optimization
   - Definition
   - Example of use

Plan

Runtime Optimization (RTO)

- RTO are basically optimization using runtime context and informations.
- Either new techniques or static techniques gone further.
- Used informations: dynamic type, data values, current state, past states, profiling information.

Attention

Optimized code tends to have many tests in them ("Am I in the optimized case ?") if not used with JIT compilation.

Plan

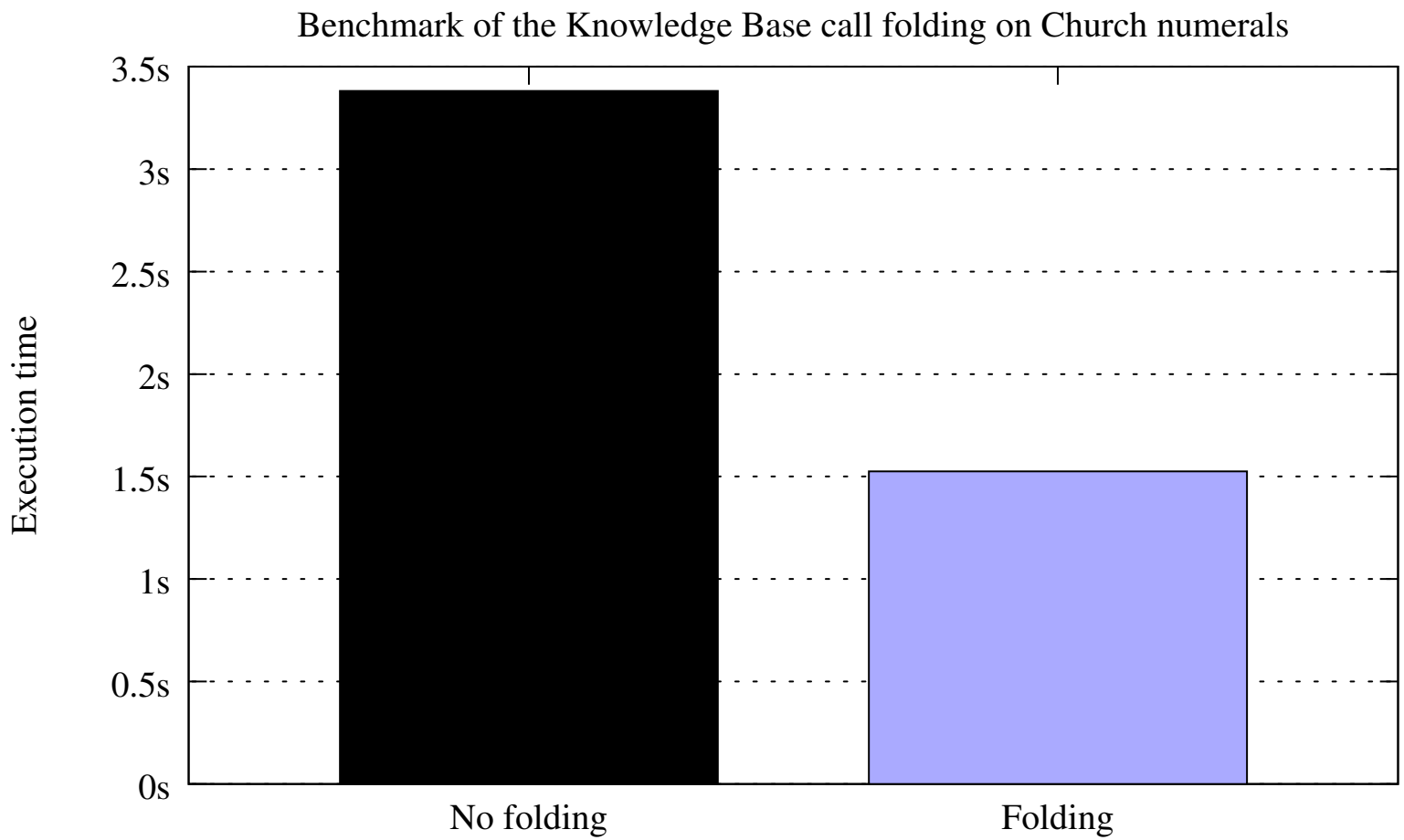
Example in Prolog

- Call to the Knowledge Base are extensive and always return the same value for a given clause.
- Many usual situations use the same clause over and over again (recursion, ...).
- Folding the returned value greatly reduces calls to the Knowledge Base with very light overhead in the worst case.

Impact on the execution time

We are using the addition of Church numerals:

```
add(X, 0, X).
add(X, succ(Y), Z) :- add(succ(X), Y, Z).
```



Other examples

- Inter-file inlining.
- On-the-fly modification of the program structure (optimize flows).

Plan

Plan

Definition and principle

- Code is compiled on-the-fly, with caching of translated code.
- Very optimized machine code (using the dynamic data type, ...).
- Slow startup: time to load and compile the code.

Plan

Examples

- HotSpot, the Oracle JVM.
- Common Language Runtime, the VM of the .Net framewok.
- PyPy, a Python interpreter and JIT compiler.

Plan

Advantages

- Once started, excellent speed (can be better than compiled code).
- More flexible than traditional compilation.
- Faster than basic interpreter.
- Very good integration of RTO: easy to alter the execution flow (inlining, folding, avoiding jump, ...)

Limitations and issues

- Heavy and complex to implement.
- Slow on startup.
- Not necessarily worth it for every language
  → Prolog execution mechanism is mainly based on its inference engine, which is already compiled.

Solutions

- Frameworks can help you (*LLVM*).
- Interpreter + JIT-compilation → only **hot spots** are JIT-compiled, rest of the program is interpreted.
- Techniques to mix the two: On-Stack-Replacement, ...

Plan

Conclusion

- Basic interpreter: very easy and fast to develop but very slow execution.
- Good performances: JIT compilation + interpreter (far more complex).
- Still many ongoing researches on various JIT-compilation and optimization policies.

- Questions ?