

Empirical Assessment of Object-Oriented Implementations with Multiple Inheritance and Static Typing

Roland Ducournau Floréal Morandat * Jean Privat

LIRMM — CNRS — Université Montpellier 2
Université du Québec à Montréal

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA

Motivation

Multiple Inheritance

- Most statically typed languages use some kind of multiple inheritance
- Doubtfull scalability

`C++` Table size is cubic in the number of classes

`Java`, `C#` Implementation of `invokeInterface` is not time-constant

Objective

- Design alternative implementations
- Evaluate their efficiency

1 Introduction

- Context
- Objectives

2 Implementation Techniques

3 Compilation Schemes

4 Test Protocol and Results

- Meta-Compiling Test Protocol
- Results and Discussion

5 Conclusion

- Prospects

Context I

Language Features

- Multiple inheritance
- Static typing

Target Languages: C++, Eiffel, Java, C#, ...

Language Independent Implementation Techniques

Three basic mechanisms

- Attribute access
- Method invocation (Late binding)
- Subtype testing

Context II

Compilation Schemes

Production of an executable from source file

- Compiler, linker, loader, ...

From pure Open World Assumption (OWA) To pure Closed World Assumption (CWA)

- Separate compilation with dynamic loading
- ...
- Global compilation

Objectives

Assessment of Runtime Efficiency

Comparing execution times depending on:

- Implementation techniques
- Compilation schemes
- Processors

With all other things being equal

Test Protocol

- Based on meta-compilation

1 Introduction

- Context
- Objectives

2 Implementation Techniques

3 Compilation Schemes

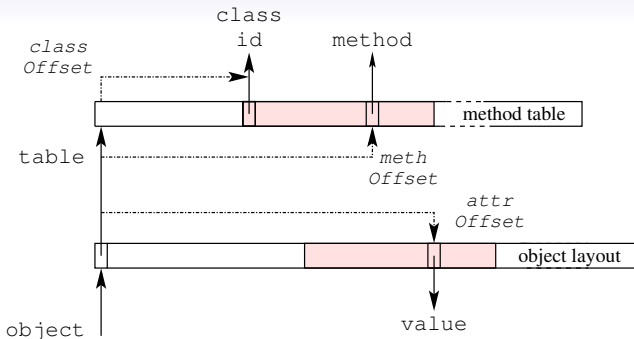
4 Test Protocol and Results

- Meta-Compiling Test Protocol
- Results and Discussion

5 Conclusion

- Prospects

Single Subtyping (SST)



Invariants

- References don't depend on their **static type**
- Positions independent of receiver's **dynamic type**
- Compatible with OWA

From SST to MI

MI can't preserve both OWA and SST Invariants

Preserving OWA

C++ subobjects (SO)

- References depend on their static types
- Overhead: Cubic table size, pointer adjustments, ...

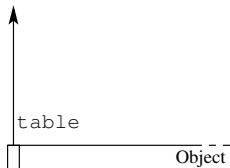
Preserving SST Invariants

Coloring

- Dixon et al. (1989), Pugh and Weddell (1990), Vitek et al (1997)
- Requires CWA at link-time
- Overhead: Holes in object layout

Perfect Hashing (PH)

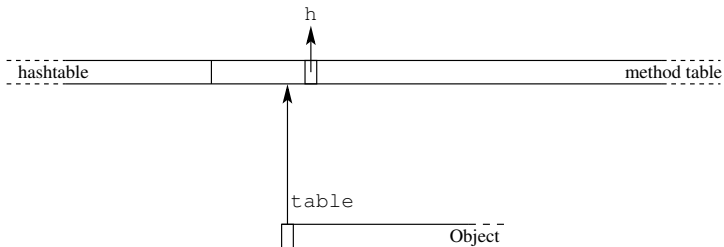
Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ Subtype testing
- `hv = Hash(h, interface Id)`
- Collision free

Perfect Hashing (PH)

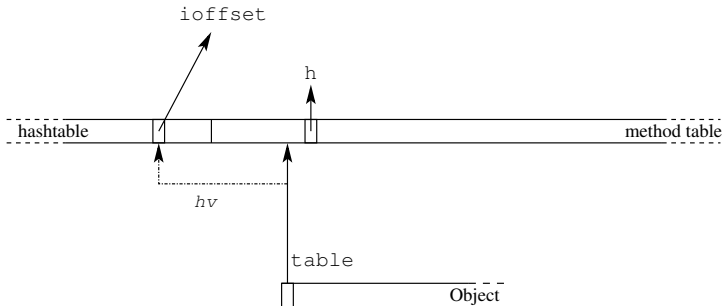
Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ Subtype testing
- $hv = \text{Hash}(h, \text{interface Id})$
- Collision free

Perfect Hashing (PH)

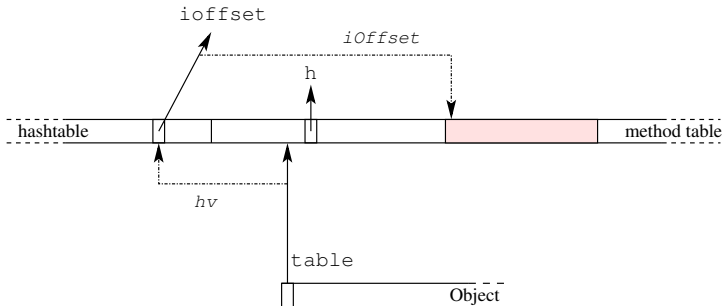
Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ Subtype testing
- $hv = \text{Hash}(h, \text{interface Id})$
- Collision free

Perfect Hashing (PH)

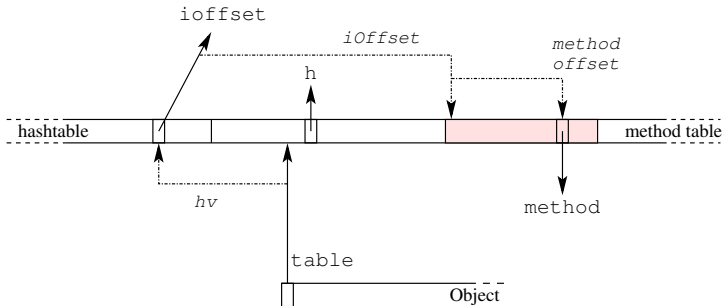
Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ `Subtype testing`
- $hv = \text{Hash}(h, \text{interface Id})$
- Collision free

Perfect Hashing (PH)

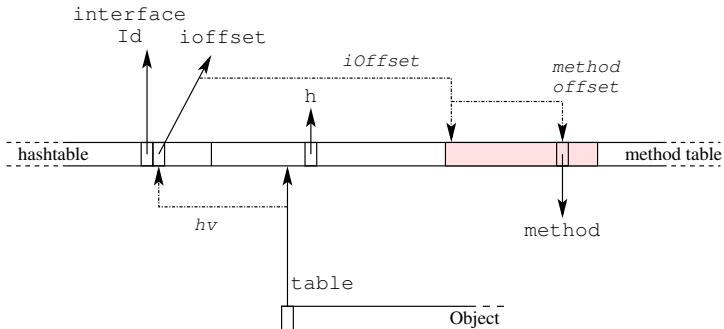
Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ Subtype testing
- $hv = \text{Hash}(h, \text{interface Id})$
- Collision free

Perfect Hashing (PH)

Alternative to C++ Subobjects



- Proposed for:
 - ▶ `invokeInterface`
 - ▶ Subtype testing
- $hv = \text{Hash}(h, \text{interface } Id)$
- Collision free

Perfect Hashing (PH)

Preserving SST Invariants

- Compatible with OWA
- Constant time
- Linear space

Hashing Functions

- Bit-wise and
- Modulo

Evaluation

- Time/Space trade-off

Binary Tree Dispatch (BTD)

Alternative to Coloring

Principle

- Tableless technique generalizing inline caches
- Type analysis / Dead code elimination \Rightarrow CWA
- Used by Smart Eiffel

Evaluation

- Logarithmic time for unbounded BTD
- $\text{BTD}_k \Rightarrow \text{depth} \leq k$ ($\text{BTD}_0 = \text{Static Calls}$)
- Complemented by **coloring** when $\text{depth} > k$
- Efficient iff k small

Caching

Principle

- It relies on some underlying implementation
- A cache is allocated into the VFT
- It memoizes last table access
- Used in production VM

Evaluation

- Code sequence markedly longer
- Efficiency depends on cache-hit rates

Increasing Cache-hit Rate

- Cache dedicated to each mechanism (empirical)
- Multiple caches with static selection (mathematical)

1 Introduction

- Context
- Objectives

2 Implementation Techniques

3 Compilation Schemes

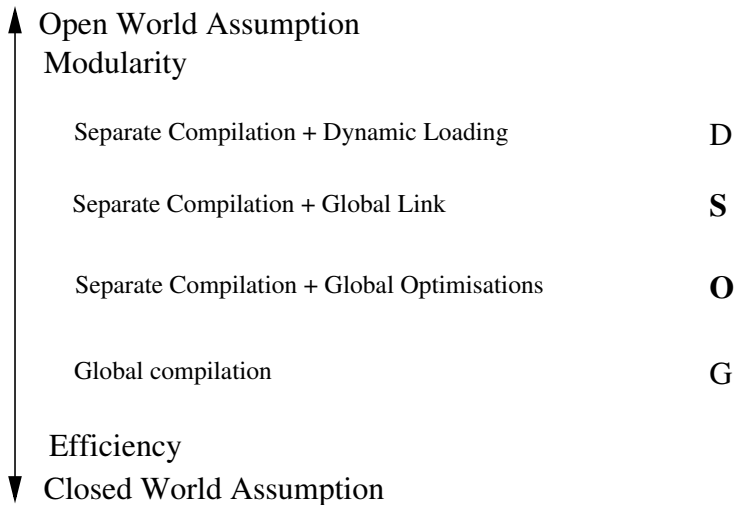
4 Test Protocol and Results

- Meta-Compiling Test Protocol
- Results and Discussion

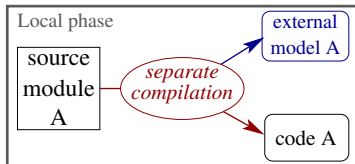
5 Conclusion

- Prospects

Compilation Schemes



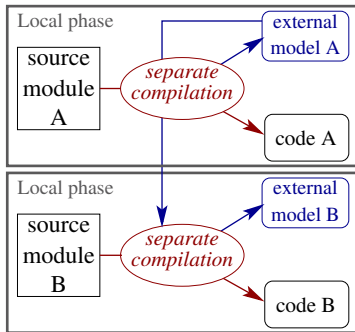
Separate Compilation with Global Linking (S)



- Modular checks
- Source code privacy

- Single subtyping efficiency

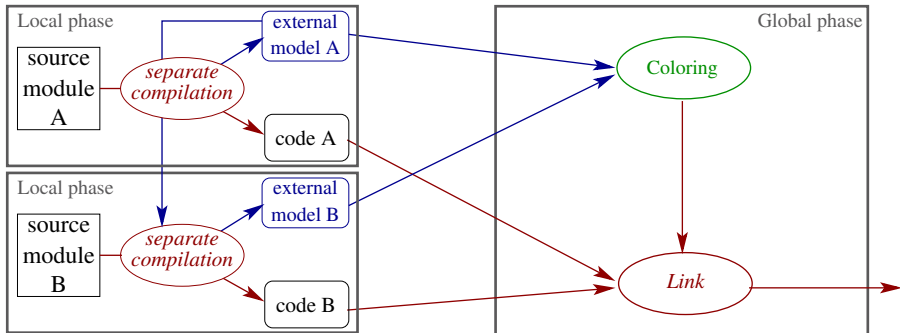
Separate Compilation with Global Linking (S)



- Modular checks
- Source code privacy

- Single subtyping efficiency

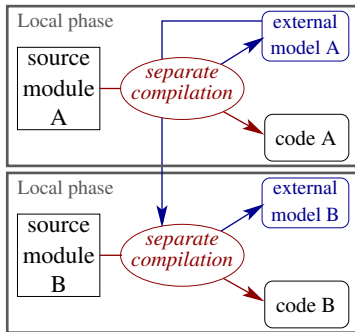
Separate Compilation with Global Linking (S)



- Modular checks
- Source code privacy

- Single subtyping efficiency

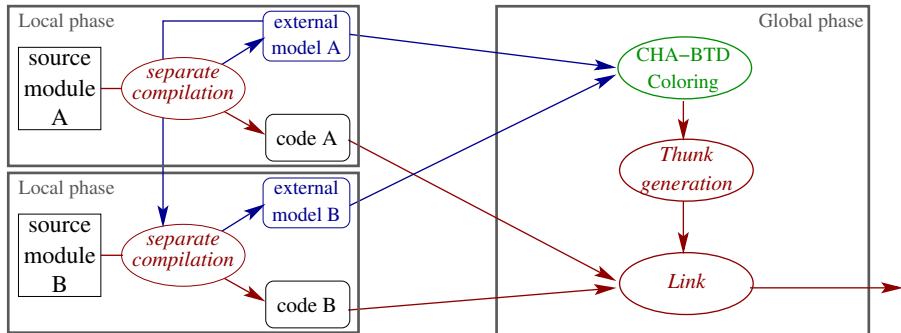
Separate Compilation with Global Optimization (O)



- Thanks for method invocation

- Monomorphic calls are static
- More optimizations available

Separate Compilation with Global Optimization (O)



- Thunks for method invocation

- Monomorphic calls are static
- More optimizations available

Implementation-Schemes Compatibility

	Dynamic	Separate	Optimized	Global
Perfect Hashing	●	*	*	*
Subobjects	◇	◇	*	*
Coloring	×	●	●	●
BTD	×	×	●	●

●: Tested

◇: Not yet tested

×: Incompatible

*: Non-Interesting

1 Introduction

- Context
- Objectives

2 Implementation Techniques

3 Compilation Schemes

4 Test Protocol and Results

- Meta-Compiling Test Protocol
- Results and Discussion

5 Conclusion

- Prospects

Test Language

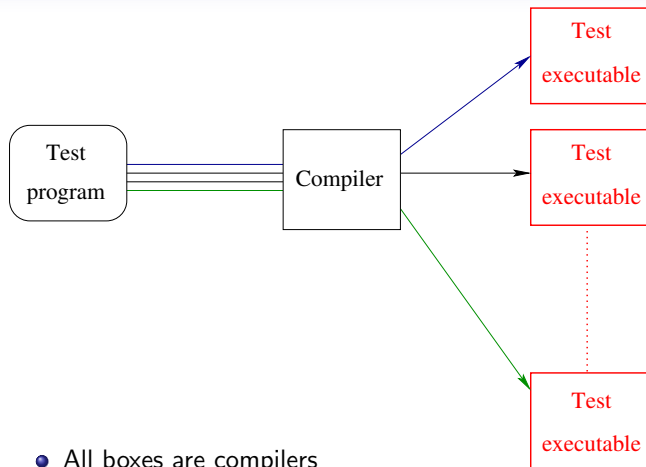
Prm the Language

- Full multiple inheritance (methods & attributes)
- Genericity
- Primitive types subtype of Object

Prmc the Compiler

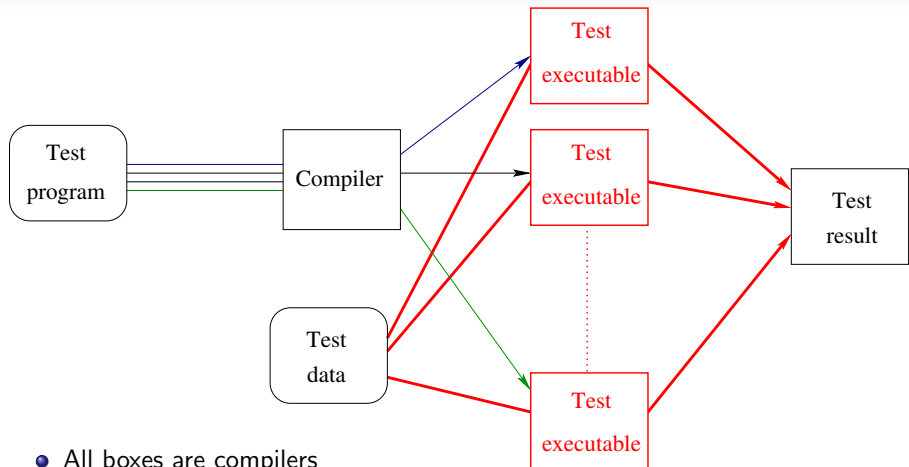
- A Prm program
- Modular
- Generate C code

Meta-Compiling Test Protocol



- All boxes are compilers
- Time measurement of the red path

Meta-Compiling Test Protocol



- All boxes are compilers
- Time measurement of the red path

Meta-Compiling Test Protocol

Runtime Reproducibility

- Deterministic code generation
 - ▶ Hashmap with predictable iteration order
 - ▶ Produces diff-equivalent binaries
- Bootstrap = actual fix point

Measurements

- Time spent by the **Prm to C** process
- Best time among several tens of runs
 - ▶ Minimises OS noise

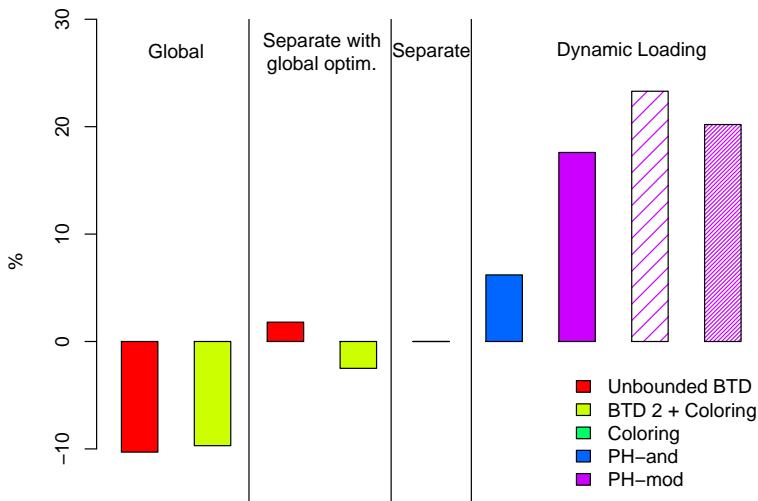
Some Statistics

Number of		Dynamic invocations
Methods calls		1720 M
BTD	0	62 %
	≤ 3	22 %
	≥ 4	16 %
Cache-hit	1	68 %
	2	71 %
	4	79 %

Consistent with statistics reported in the literature

Runtime Efficiency

Intel Core2 E8500



Evaluation

Compilation schemes

- Global scheme is far better than separate
- Optimized scheme slightly better than separate
- Dynamic loading is very expensive especially in full Multiple Inheritance

Implementation techniques

- BTD+Coloring optimal
- PH-and efficient for Java interfaces (by extrapolation)
- Caching inefficient even with PH-mod

1 Introduction

- Context
- Objectives

2 Implementation Techniques

3 Compilation Schemes

4 Test Protocol and Results

- Meta-Compiling Test Protocol
- Results and Discussion

5 Conclusion

- Prospects

Conclusion

First systematic comparisons

- Language independent

between

- Implementation techniques
- Compilation schemes
- Processors

ceteris paribus (with all other things being equal)

- Mainly confirm previous theoretical results
- Significant variation according to processors (about ten tested) but similar behaviours

Conclusion

Prm the testbed

- Modular compiler open to new implementations and schemes
- **Repeatable** and **reproducible** tests
- Single program tested, but intensive OO mechanism usage

Prm the languages

- **Prm** : Dedicated to test

<http://www.lirmm.fr/prm/>

- **Nit** : **User friendly language (recommended)**

<http://www.nitlanguage.org/>

Prospects

Testbed extension

- Other implementations (C++ subobjects, ...)
- Other processors and architectures
- Other metrics (cpu cache misses, memory usage, ...)
- Heterogeneous vs homogeneous genericity
- Other optimisations (garbage collector, ...)

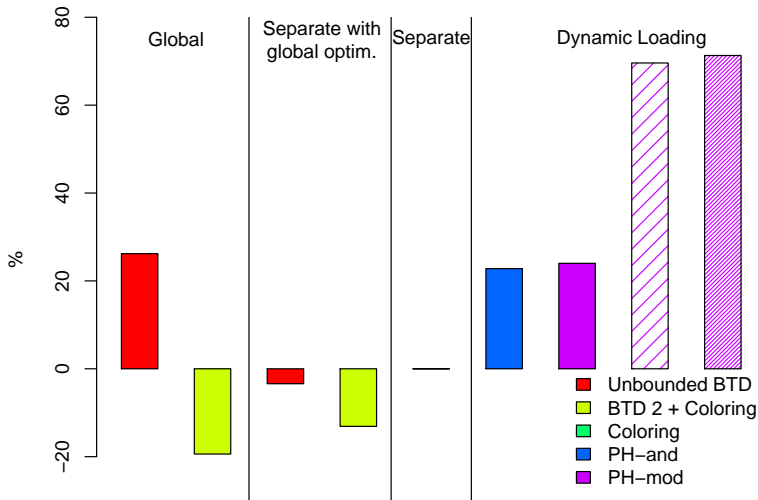
Virtual Machine application

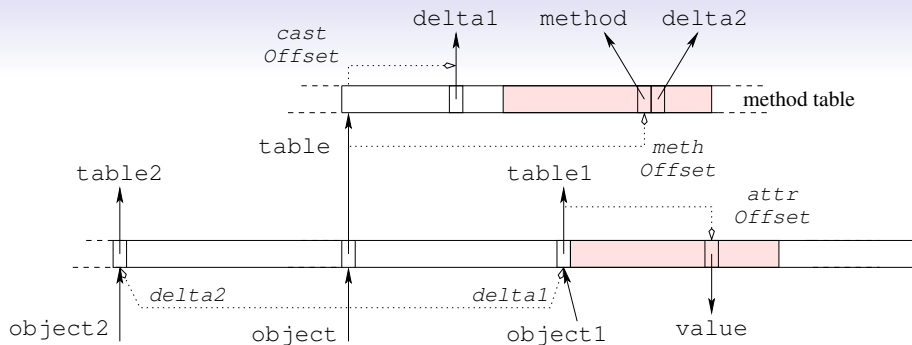
- Perfect Hashing on Production VM
- Application of link-time global optimization to adaptive compilers (JIT)
- Full multiple inheritance VM (as efficient as Java/.NET)

Thanks ...

Executable Size

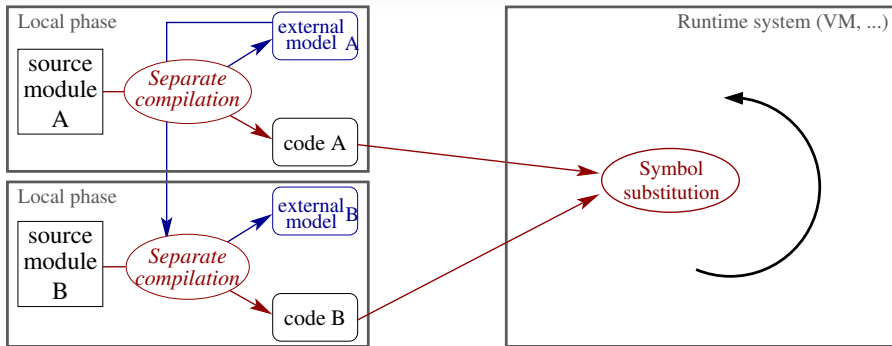
Intel Core2 E8500





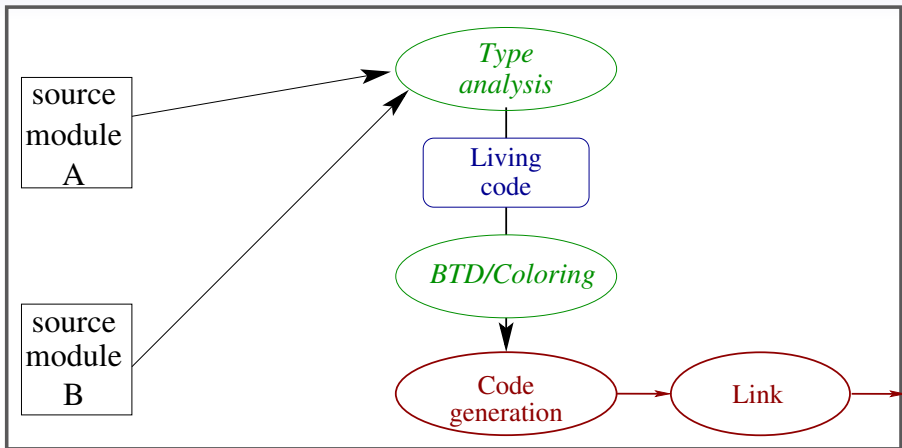
- Reference depend on it's static types
- Cubic table size in the number of classes
- Pointer adjustments

Separate Compilation and Dynamic Loading (D)



- Pure OWA
- Source code privacy
- Modular checks
- Fast recompilation
- **No optimization available without recompilations**

Global Compilation (G)



- Lots of optimizations available
- More compact code
- No modular checks
- Heavy recompilation