Four Steps to the Separate Compilation of Modelica

Equation-Based Object Oriented Modeling Languages and Tools 2010, Oslo

Christoph Höger, Peter Pepper, Florian Lorenzen

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
Fraunhofer FIRST

2010/10/03

Outline I

1. Motivation
2. Step one: Runtime instantiation
3. Step two: Coercions
4. Step three: Resolve dynamic binding
5. Step four: cover more language features (expandable connectors etc.)

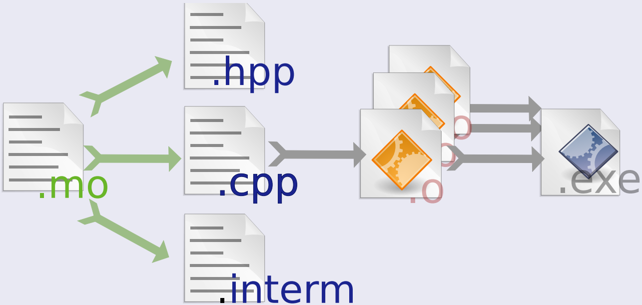
Goal

What is separate compilation?

- Translation of a *single* compilation unit (a.k.a. source file) at once
- Generation of *reusable* output files (a.k.a. object files)
- Distribution of *partial* compilation results with well defined interfaces (a.k.a. libraries)
- Integration into existing build systems (e.g. GNU make)

Compilation Scheme

Compilation of a Modelica compilation unit



Compile time instantiation

Every (current) Modelica Compiler is a Modelica Interpreter

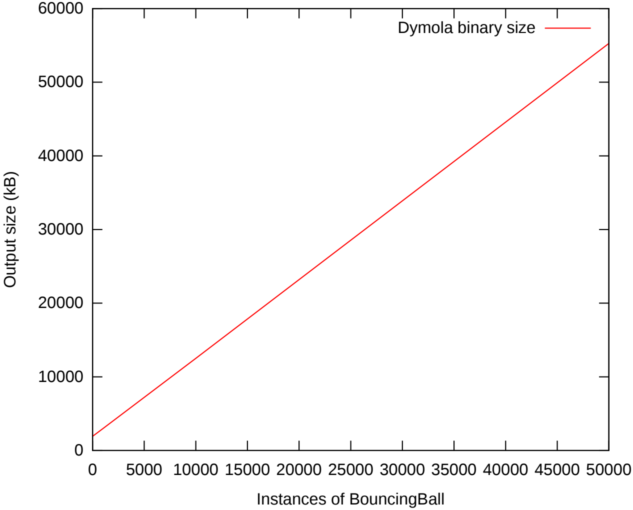
Compile-time Flattening . . .

- . . . effectively means interpreting a functional language
- . . . instantiates *every* object
- . . . creates the same expressions again and again
- . . . prevents: Structural dynamics, dynamic Arrays etc.

Problem!

Modelica Turing-complete → Compiler might not terminate!

Space Waste


Dymola binary size
Output size (kB)
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
Instances of BouncingBall

Doing it the other way around

Instantiation after compilation!

With this, Modelica has a operational semantics like any other OO language.

Modelica Abstract Machine

Modelica Abstract Machine - Sketch

- Create a runtime type for Equations and Variables
- Compile every Model into a functon $f : ([Equation], [Variable])$
- Add class parametrization: $f : \alpha \rightarrow ([Equation], [Variable])$
- Formally define the instantiation of models by function evaluation

MAM: Benefits and Drawbacks

Benefits

MAM code . . .

- . . . Can be compiled separately for *every* model
- . . . Can be used either in a Modelica Linker or in a Interpreter
- . . . Can be redistributed
- . . . Can be JIT compiled
- . . . Gives Model Structural Dynamics semantics *for free*

Drawbacks

MAM code . . .

- **Cannot** be compiled into efficient code directly (needs causalization etc.)
- May cause runtime exceptions

Structural Subtyping

"If it walks like a duck, and talks like a duck, it is a duck!"

Relevant for separate compilation

- Target language has usually a nominal type system (C++), or no type system at all (C)
- The fields of e.g. a struct are determined upon compilation
- How can the compiler know how to access a type's element that may not even exist yet?
- $\rightarrow$ of course it cannot.

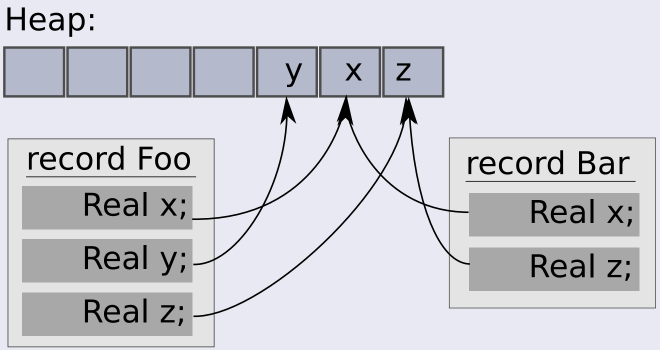
Coercions

Coercion functions

- General idea: let the callee handle subtyping issues
- Callee knows what type is needed from typecheck!
- If instead of type A an object of type B is expected, create and use coercion function $C_{A \to B} : A \to B$
- Has performance impact only, if objects change (if used with references).
- $\to$ Ideal for Modelica
- Works also for function and class parameters

Scheme

Coercions with pointers



inner/outer

Definition

inner/outer are Modelica's dynamic binding notation

Compilation Unit Planet.mo

```
model Planet
    inner Real g;
    BouncingBall Ball;
    equation
        g = -9.81;
end Planet;
```

Compilation Unit BouncingBall.mo

```
model BouncingBall
    outer Real g;
    ...
```

inner/outer

Problem:

How to compile with outer definitions, before the inner definition can be known?

dynamic binding as reference passing

If g was a parameter

```
model Planet
    parameter Real g = -9.81;
    BouncingBall Ball(g = g);
end Planet
```

Solution:

inner/outer means passing a (coerced) reference to submodels!

Conclusion:

Switch MAM to parameter evaluation by reference

Compilation output

BouncingBall.cc

```
ptrBouncingBall createNewVariableInstance(Runtime& runtime
, ptrReal g) {
   ptrBouncingBall instance(new _BouncingBall::data());
   instance->g = g;
   instance->e = _Real::createNewConstInstance(runtime );
...
```

Planet.cc

```
ptrPlanet createNewVariableInstance(Runtime& runtime ) {
   ptrPlanet instance(new _Planet::data());
   instance->g = _Real::createNewVariableInstance(runtime);
   instance->ball =
_BouncingBall::createNewVariableInstance(runtime,
instance->g);
...
```

Covering Modelica completely

What is left

- Modelica is a very complex beast
- Try to reduce special cases and semantics to the above mentioned cases
- Should, in general, suffice

Example: expandable connectors

Problem

- every connect() equation expands the connector, if neccessary
- again, information at compile time is incomplete
- the type of the actual connector is only known after linking
- But: We don't need it beforehand!

Solution:

An expandable connector can be written as a type parameter with a default value.

Example: expandable connectors

expanding a connector:

```
Real x;
expandable connector conn;
equation
  connect(conn, x);
```

becomes:

```
Real x;
replaceable class connStruct;
outer connStruct conn;
equation
  conn.x = x;
```

Conclusion

Conclusion

- Modelica as a Language can be compiled separately
- How to make use of that fact is a tooling issue

Thank you! Any questions?