Overview: Sequential compilation and other models

David Pineau

December 9, 2011

Plan

1. Introduction
   - What is “sequential compilation” ?
   - Why this talk ?

What is “sequential compilation” ?

Sequential compilation is a model of code generation where the code produced presents the same flow of execution as the source code. This can be done by simple translation steps between source and target code.

Why this talk ?

- Use of Rathaxes's development: another kind of language, need for another model of generation
- Describe a model out of the ordinary for new purposes

Plan

2. Examples of sequential compiler
   - Source and Target languages
   - Strong points
   - Weaknesses

Source and Target languages

Some examples:

- C
- Java
- So many others...

Code Compilation steps (for GCC/G++):

1. C
2. Intermediate language (as an AST)
3. asm
4. machine

Let's compare those languages

C is an *imperative* language with *low* expressivity.

Asm is an *imperative* language with *no* expressivity.

Translation: How is it done ?

- "Machine description" files (*.md*)
- Translating the source code into an intermediate language
- Using the *.md* files to translate an intermediate instruction into an asm instruction
- Translate the generated asm into machine code

Strong Points

1. Relatively easy to add support for a new architecture
2. Allows optimizations
3. Straight-forward translation method

Weaknesses

1. Mostly applicable when source/target languages are similar in terms of paradigm
2. Dependant on the target language (the *.md* files)

Plan

3. Examples of a non-sequential compiler
   - Source and Target language
   - Strong points
   - Weaknesses

Source and Target language

Some examples:

- Rathaxes (Domain Specific Language)
- SQL

A source language in three parts (Rathaxes is a DSL):

- Front-End (Pure DSL part)
- Middle-End (interfaces for type enforcement)
- Back-End (templates of instrumentalized code in the target language: C)

The numerous compilation steps

Middle-End compilation steps:

1. Parsing
2. Type-checking (auto-validation with the other interfaces)
3. Registration in a cache (prevents having to re-parse and check everything everytime).

The numerous compilation steps

Back-End compilation steps:

1. Parsing
2. Type-checking (validate the types and implementation against the interfaces' description)
3. Instrumentalization resolver generation (to be loaded at front-end compilation time)
4. Registration in a cache

The numerous compilation steps

Front-End compilation steps:

1. Parsing
2. Type-checking (validate that nothing is missing, and that the implementation respects the interfaces' constraints)
3. Load the necessary templates for generation and recursively resolve them with the help of the generated code
4. Generation of the target code

Strong points

- Strongly appropriated to DSL compilation
- Generic model of generation, easily reusable
- Adapted to an evolutive language in an evolutive context (case of *Rathaxes*)

Weaknesses

- Quite complex, with many steps
- Dependant on the target language (back-end templates)

Plan

4. Sequential and non-sequential comparison
   - Similarities
   - Differences

Similarities

- Dependant on the target language for the evolutive concepts
- Use of templates for either architecture support or code generation

Differences

	Sequential	Non-Sequential
Type of language	general-purpose	DSL
Complexity	Simple	Complex
Optimization	Available	None

Plan

5 Choosing your generation model
- Analysis of the source and target languages
- Needs for evolution

Analysis of the source and target languages

- How could you qualify the level of abstraction of the source language ?
- Are both languages (Source and Target) of a similar paradigm, level of abstraction ?
- Is it easy to get from the source paradigm to the target paradigm ?

Needs for evolution

- Must the language be supple and support an evolutive back-end ?
- Are the source and target languages fixed ?
- Must it be easy to add a new back-end implementation ?

Thanks

Questions ?