Automated Code Generation using Dynamic Programming Techniques

Igor Boehm

Supervisor: Prof. Dr. Dr. h.c. Hanspeter Mössenböck
Outline

Motivation
Where and What is it good for?

Theoretical Preliminaries
Code Generation Example
Dynamic Programming
Tree Pattern Matching using Dynamic Programming

Tree Pattern Matching Language
Language Specification

Finis
Current Status
Why do we need Code Generation Tools?

- Writing a good compiler for a modern programming language is not easy...
Where and What is it good for?

Tame Complexity through Automation!

...so we like to let tools figure out how to do the boringly mechanical parts, and only fill in the relevant bits and pieces.
**Motivation**

Theoretical Preliminaries

Tree Pattern Matching Language

**Where and What is it good for?**

Front End

Parser

CSA

Optimizer

Analysis

IR

IR

IR

Select

Schedule

Allocate

Backend Generator

Tree Pattern Matching Engine

Tree Pattern Matching is the key!

AST Patterns

Mapping between IR & Machine Description

Backend Generator

Note similarities and differences to Parser Generators!

Semantic Actions

**Igor Boehm**

Johannes Kepler University Linz, Austria

Automated Code Generation using Dynamic Programming Techniques
What do we actually want to do?

- Given an IR in the form of an AST:
  - We want the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the essential part of the problem, namely the task of generating code for AST trees and subtrees.
  - We do not want:
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
Motivation

Theoretical Preliminaries

Tree Pattern Matching Language

Finis

What do we actually want to do?

- Given an IR in the form of an AST:
  - We **want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.
  - We **do not want**:
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:
  - **We want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.
  - **We do not want:**
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:
  - We **want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.
  - We **do not want**:
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:
  - **We want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.

- **We do not want:**
  - to clutter our code with boring tree traversals
  - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:

  - We **want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.

  - We **do not want**:
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:
  - *We want* the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.
  - *We do not want*:
    - to clutter our code with boring tree traversals
    - to worry about coding the details of optimal instruction selection
What do we actually want to do?

- Given an IR in the form of an AST:
  - We **want** the following:
    - obviously we want to generate Machine Code
    - but the generated code should be optimal!
    - concentrate on the *essential* part of the problem, namely the task of generating code for AST trees and subtrees.

- We **do not want**:
  - to clutter our code with boring tree traversals
  - to worry about coding the details of optimal instruction selection
Lets say we have an AST for the following operation and want to emit code for it:

\[ w := x - 2 \ast y \]
If we were to code this manually, we would probably do something like the following:

Each tile corresponds to a sequence of operations.

If those operations are emitted in the appropriate order they *implement* the tree.
So what’s hard about this?

- Well, the hard part is to find the optimal set of tiles to tile the tree.

- In order to see why that is a problem we will *connect* tiles to AST subtrees by:
  - providing a set of *rewrite rules*
  - associate *code templates* with rewrite rules
## Rewrite Rules with Code Templates for Abstract Syntax Tree

<table>
<thead>
<tr>
<th>Rewrite Rule</th>
<th>Code Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Goal → Assign</td>
<td><strong>store r₂ → r₁</strong></td>
</tr>
<tr>
<td>2 Assign → (Reg₁, Reg₂)</td>
<td><strong>store r₂ → r₁</strong></td>
</tr>
<tr>
<td>3 Reg → LAB₁</td>
<td><strong>loadI l₁ → r_{new}</strong></td>
</tr>
<tr>
<td>4 Reg → VAL₁</td>
<td><strong>load n₁ → r_{new}</strong></td>
</tr>
<tr>
<td>5 Reg → NUM₁</td>
<td><strong>load n₁ → r_{new}</strong></td>
</tr>
<tr>
<td>6 Reg → REF(Reg₁)</td>
<td><strong>load r₁ → r_{new}</strong></td>
</tr>
<tr>
<td>7 Reg → REF(+ (Reg₁, Reg₂))</td>
<td><strong>loadAO r₁, r₂ → r_{new}</strong></td>
</tr>
<tr>
<td>8 Reg → REF(+ (Reg₁, NUM₂))</td>
<td><strong>loadAI r₁, n₂ → r_{new}</strong></td>
</tr>
<tr>
<td>9 Reg → REF(+ (LAB₁, Reg₂))</td>
<td><strong>loadAI r₂, l₁ → r_{new}</strong></td>
</tr>
<tr>
<td>10 Reg → +(Reg₁, Reg₂)</td>
<td><strong>add r₁, r₂ → r_{new}</strong></td>
</tr>
<tr>
<td>11 Reg → +(Reg₁, NUM₂))</td>
<td><strong>addI r₁, n₂ → r_{new}</strong></td>
</tr>
<tr>
<td>12 Reg → +(LAB₁, Reg₂))</td>
<td><strong>addI r₂, l₁ → r_{new}</strong></td>
</tr>
</tbody>
</table>

**Table:** Modified example from *Cooper & Torczon p.561*
Given the previous rewrite rules for our AST, let's consider **Tile 3** from our example and try to tile it:

Initial AST

Matches 3

Matches 8

Reduced
But there is a plethora of rewrite sequences (or tilings) for this trivial subtree!!!
How do we select the optimal sequence of rewrite rules?

- We need some metric which enables us to compare various selections of rewrite rules with each other.
  - Solution: Annotate rewrite rules with costs!

- We need some clever algorithm to sort out the optimal rewrite sequence.
  - Solution: Dynamic Programming will do the trick!
How do we select the optimal sequence of rewrite rules?

- We need some metric which enables us to compare various selections of rewrite rules with each other.
  - Solution: Annotate rewrite rules with costs!

- We need some clever algorithm to sort out the optimal rewrite sequence.
  - Solution: Dynamic Programming will do the trick!
How do we select the optimal sequence of rewrite rules?

We need some metric which enables us to compare various selections of rewrite rules with each other.

**Solution:** Annotate rewrite rules with costs!

We need some clever algorithm to sort out the optimal rewrite sequence.

**Solution:** Dynamic Programming will do the trick!
How do we select the optimal sequence of rewrite rules?

- We need some metric which enables us to compare various selections of rewrite rules with each other.
  - Solution: *Annotate* rewrite rules with **costs**!

- We need some clever algorithm to sort out the optimal rewrite sequence.
  - Solution: *Dynamic Programming* will do the trick!
How do we select the optimal sequence of rewrite rules?

We need some metric which enables us to compare various selections of rewrite rules with each other.

**Solution:** *Annotate* rewrite rules with *costs*!

We need some clever algorithm to sort out the optimal rewrite sequence.

**Solution:** *Dynamic Programming* will do the trick!
Dynamic Programming is a method of solving problems exhibiting the following properties:

- the approach for a given problem assumes a recursive solution, with a bottom-up evaluation of the solution.

- sub-solutions can be recorded (e.g. in a table) for reuse.
Dynamic Programming is a method of solving problems exhibiting the following properties:

- the approach for a given problem assumes a recursive solution, with a bottom-up evaluation of the solution.
- sub-solutions can be recorded (e.g. in a table) for reuse.
Dynamic Programming is a method of solving problems exhibiting the following properties:

- the approach for a given problem assumes a *recursive solution*, with a *bottom-up* evaluation of the solution.
- sub-solutions can be recorded (e.g. in a table) for *reuse*. 
One optimal solution for our previous tiling problem looks as follows:

Recursively work bottom up reusing recorded results.
So we are almost there!

- We know how to optimally tile an AST using Dynamic Programming.

- We know how to specify rewrite rules (also called tree patterns or productions).

- We know how to specify code templates (semantic actions) for AST patterns.
So we are almost there!

- We know how to optimally tile an AST using Dynamic Programming.

- We know how to specify rewrite rules (also called tree patterns or productions).

- We know how to specify code templates (semantic actions) for AST patterns.
So we are almost there!

- We know how to optimally tile an AST using Dynamic Programming.
- We know how to specify rewrite rules (also called tree patterns or productions).
- We know how to specify code templates (semantic actions) for AST patterns.
Tree Pattern Matching using Dynamic Programming works as follows:

- Two passes over the AST:
  - Pass 1: Finds the optimal tiling of an AST using Dynamic Programming.
  - Pass 2: Emits semantic actions (code templates) for tiled AST.
Tree Pattern Matching using Dynamic Programming works as follows:

- **Two passes over the AST:**
  - **Pass 1:** Finds the optimal tiling of an AST using Dynamic Programming.
  - **Pass 2:** Emits semantic actions (code templates) for tiled AST.
Tree Pattern Matching using Dynamic Programming works as follows:

- Two passes over the AST:
  - **Pass 1:** Finds the optimal tiling of an AST using Dynamic Programming.
  - **Pass 2:** Emits semantic actions (code templates) for tiled AST.
Tree Pattern Matching using Dynamic Programming works as follows:

- *Two* passes over the AST:
  - **Pass 1**: Finds the optimal tiling of an AST using Dynamic Programming.
  - **Pass 2**: Emits semantic actions (code templates) for tiled AST.
Let’s look at where our code generator is used in the compiler design process:

- **Tree Pattern grammar spec.** for our AST
  - ...takes spec. and produces...
  - Java, C#, ..., code which can be plugged into our Compiler
- **Front End**
  - **Scan** ➔ **Parse** ➔ **CSA**
- **Optimizer**
  - **Opt 1** ➔ **Opt 2** ➔ ... ➔ **Opt n**
  - **Analysis**
  - **IR** ➔ **Scan** ➔ **Opt 1** ➔ **Opt 2** ➔ ... ➔ **Opt n**
- **Back End**
  - **Schedule** ➔ **Allocate**
Tree Pattern Matching Language

- General Structure of a Grammar Specification:
  - import, include statements
  - global variable definitions
  - methods and functions
  - rewrite rules
  - cost
  - semantic action
  - tree pattern

Igor Boehm
Johannes Kepler University Linz, Austria

Automated Code Generation using Dynamic Programming Techniques
We have the following Node definition together with an example Abstract Syntax Tree:
Example 1

```plaintext
generator  -- Import Statements
( : import java.util.LinkedList;
    import codegen.Reg;
    import Node;
    :
)

declarations  -- General Declarations
( : public LinkedList instrList = new LinkedList(); :
)

operators  -- Operators (Our Node Kinds)
CONST(: E_CONST:), ADD(: E_ADD:),
SUB(: E_SUB:), MULT(: E_MULT:)

rules  -- Production Rules
reg = CONST c1 (: c1.result = Reg.getNextReg();
    instrList.add("loadI " +
        c1.val + "," +
        c1.result);
    :
): 1

| ADD a1 ( reg r1, reg r2 )
| (: a1.result = Reg.getNextReg();
|    instrList.add("add " +
|        r1.result + "," +
|        r2.result + "," +
|        a1.result);
|    :
|): 1

| SUB s1 ( reg r1, reg r2 )
| (: s1.result = Reg.getNextReg();
|    instrList.add("sub " +
|        r1.result + "," +
|        r2.result + "," +
|        s1.result);
|    :
|): 1

| MULT m1 ( reg r1, reg r2 )
| (: ...
|    :
|): 2

end
```

Igor Boehm

Automated Code Generation using Dynamic Programming Techniques

Johannes Kepler University Linz, Austria
While the previous example looks nice at first sight, it is still quite cumbersome:

- **Problem 1:** The way in which result registers are returned is by abusing the node class!

- **Problem 2:** The list holding instructions is global!

- We can do much better than this!!!
While the previous example looks nice at first sight, it is still quite cumbersome:

- **Problem 1:** The way in which result registers are returned is by abusing the node class!

- **Problem 2:** The list holding instructions is global!

- We can do much better than this!!!
While the previous example looks nice at first sight, it is still quite cumbersome:

- **Problem 1**: The way in which result registers are returned is by abusing the node class!
- **Problem 2**: The list holding instructions is global!

We can do much better than this!!!
While the previous example looks nice at first sight, it is still quite cumbersome:

- **Problem 1**: The way in which result registers are returned is by abusing the node class!
- **Problem 2**: The list holding instructions is global!

We can do much better than this!!!
We can use the **Attribute Grammar** formalism to correct our previous ’flaws’:

- **Solution to Problem 1**: The register production produces a *register* as its output and takes a list of *instructions* as its input to append its instructions to.

- **Solution to Problem 2**: The instruction list is passed as a parameter and thus must not be declared as a ’global’ class variable.

Let’s look at an example for clarification...
We can use the **Attribute Grammar** formalism to correct our previous ’flaws’:

- **Solution to Problem 1**: The register production produces a *register* as its output and takes a list of *instructions* as its input to append its instructions to.

- **Solution to Problem 2**: The instruction list is passed as a parameter and thus must not be declared as a ’global’ class variable.

- Let’s look at an example for clarification...
We can use the **Attribute Grammar** formalism to correct our previous ’flaws’:

- **Solution to Problem 1**: The register production produces a *register* as its output and takes a list of *instructions* as its input to append its instructions to.

- **Solution to Problem 2**: The instruction list is passed as a parameter and thus must not be declared as a ’global’ class variable.

Let’s look at an example for clarification...
We can use the **Attribute Grammar** formalism to correct our previous ’flaws’:

- **Solution to Problem 1**: The register production produces a *register* as its output and takes a list of *instructions* as its input to append its instructions to.

- **Solution to Problem 2**: The instruction list is passed as a parameter and thus must not be declared as a ’global’ class variable.

- Let’s look at an example for clarification...
Example 2

```java
generator -- Import Statements
C: import java.util.LinkedList;
    import comp.codegen.Reg;
    import comp.ast.Node;
    : )
declarations -- General Declarations (not needed now)
operators -- Operators (our Node Kinds)
    CONST(:E_CONST:), ADD(:E_ADD:),
    SUB(:E_SUB:), MULT(:E_MULT:)
rules
    -- Production rules for registers:
    -- @out: produce a result register
    -- @in: list to add instructions to
    reg <; out String reg, List instr > :
    -
    | CONST c1 C:
    |    reg = Reg.getNextReg();
    |    instr.add("loadI " +
    |    c1.val + "," +
    |    reg);
    | : 1
    | ADD ( reg <; out r1, instr > ,
    |      reg <; out r2, instr > )
    | C:
    |    reg = Reg.getNextReg();
    |    instr.add("add " +
    |    r1 + "," +
    |    r2 + "," +
    |    reg);
    | : 1
    | SUB ( reg <; out r1, instr > ,
    |      reg <; out r2, instr > )
    | C:
    |    reg = Reg.getNextReg();
    |    instr.add("sub " +
    |    r1 + "," +
    |    r2 + "," +
    |    reg);
    | : 1
    | -- MULT pattern omitted
    |    :
    | end
```

Automated Code Generation using Dynamic Programming Techniques
The simple example used up until now doesn’t really convey the full power of the Tree Pattern Matching Language, namely:

- Arbitrary nesting of patterns.
- The ability to *sprinkle* semantic actions almost anywhere.

...those features will be revealed during my last presentation where I will demonstrate a complete working example.
The simple example used up until now doesn’t really convey the full power of the Tree Pattern Matching Language, namely:

- Arbitrary nesting of patterns.
- The ability to *sprinkle* semantic actions almost anywhere.
- ...those features will be revealed during my last presentation where I will demonstrate a complete working example.
The simple example used up until now doesn’t really convey the full power of the Tree Pattern Matching Language, namely:

- Arbitrary nesting of patterns.
- The ability to *sprinkle* semantic actions almost anywhere.

...those features will be revealed during my last presentation where I will demonstrate a complete working example.
The simple example used up until now doesn’t really convey the full power of the Tree Pattern Matching Language, namely:

- Arbitrary nesting of patterns.
- The ability to *sprinkle* semantic actions almost anywhere.

...those features will be revealed during my last presentation where I will demonstrate a complete working example.
Current Status

What has been done up until now?

- An extensive prototyping phase helped to identify the core features of the tree pattern matching language by trying to solve real world problems with it.

- Before any implementation started a semantics for the language has been specified in terms of a denotational semantics and semantic algebras.

- A fully functional Lexer and Parser has already been implemented.
Current Status

What has been done up until now?

- An extensive prototyping phase helped to identify the core features of the tree pattern matching language by trying to solve real world problems with it.
- Before any implementation started a semantics for the language has been specified in terms of a denotational semantics and semantic algebras.
- A fully functional Lexer and Parser has already been implemented.
What has been done up until now?

- An extensive prototyping phase helped to identify the core features of the tree pattern matching language by trying to solve real world problems with it.

- Before any implementation started a semantics for the language has been specified in terms of a denotational semantics and semantic algebras.

- A fully functional Lexer and Parser has already been implemented.
What has been done up until now?

- An extensive prototyping phase helped to identify the core features of the tree pattern matching language by trying to solve real world problems with it.

- Before any implementation started a semantics for the language has been specified in terms of a denotational semantics and semantic algebras.

- A fully functional Lexer and Parser has already been implemented.
Current Status

What must still be done?

- Implement good context sensitive analysis to ease development of Tree Pattern Matching grammar specifications.
- Finally, emit code for the given specifications!
What must still be done?

- Implement good context sensitive analysis to ease development of Tree Pattern Matching grammar specifications.
- Finally, emit code for the given specifications!
What must still be done?

- Implement good context sensitive analysis to ease development of Tree Pattern Matching grammar specifications.
- Finally, emit code for the given specifications!
Thank you for your attention!