Templates and friends

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
What’s a template?
"Hello World" with StringTemplate in Java

```java
import org.stringtemplate.v4.*;
...
ST hello = new ST("Hello, <name>");
hello.add("name", "World");
System.out.println(hello.render());
```

[String template with one parameter](http://www.antlr.org/wiki/display/ST4/Introduction, 10 December 2012)
"Hello World" with StringTemplate in C#

```csharp
using Antlr4.StringTemplate;

Template hello = new Template("Hello, <name>");
hello.Add("name", "World");
Console.Out.WriteLine(hello.Render());
```

"Hello World" with StringTemplate in Python

```python
import stringtemplate4

hello = stringtemplate4.ST("Hello, <name>")
hello["name"] = "World"
print str(hello.render())
```

A template is a declarative, parametrized, and executable description of a text generator.

Find a proper definition that fits into the context of this lecture.
Motivation:

templates in web development
Web programming with PHP

HTML with embedded PHP executed on the web server

```php
<html>
<head>
<title>HelloWorld WebApp</title>
</head>
<body>
<?php echo '<p>Hello World!</p>'; ?></body>
</html>
```

Resulting HTML rendered in the web browser

Hello World!
Web programming with Rails
Web programming with Rails

```html
<div class="headline"><h2>101companies Ruby on Rails Web App</h2></div>

```
Web programming with Struts/JSP

...
"Model to Text" with JET

```html
<%@ jet package="generator.website" class="Simple2HTML" imports ="java.util.Iterator org.eclipse.emf.common.util.EList datamodel.website.*;" %>

<% Webpage website = (Webpage) argument; %>
<html>
  <head>
    <title><%=website.getTitle()%></title>
    <meta name="description" content="<%=website.getDescription()%>">
    <meta name="keywords" content="<%=website.getKeywords()%>">
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>List of Articles Names</h1>
    <%EList<Article> list = website.getArticles();
    for (Iterator<Article> iterator = list.iterator(); iterator.hasNext();)
      Article article = iterator.next();%>
      <b><%=article.getName()%></b> <br>
    <%}%>
  </body>
</html>
```

Similar to JSP. Works on top of EMF.

Generative (MDE) approach such that a designated class is generated from the template.

http://www.vogella.com/articles/EclipseJET/article.html

import generator.website.Simple2HTML;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import datamodel.website.MyWeb;
import datamodel.website.Webpage;

public class SimpleWebTest {
    private static String TARGETDIR = "./output/";

    public static void main(String[] args) {
        EMFModelLoad loader = new EMFModelLoad();
        MyWeb myWeb = loader.load();
        createPages(myWeb);
    }

    private static void createPages(MyWeb myWeb) {
        ...
    }

    http://www.vogella.com/articles/EclipseJET/article.html

...  
private static void createPages(MyWeb myWeb) {
    Simple2HTML htmlpage = new Simple2HTML();
    FileWriter output;
    BufferedWriter writer;
    for (Iterator<Webpage> iterator = myWeb.getPages().iterator(); iterator.hasNext();)
    {
        Webpage page = iterator.next();
        System.out.println("Creating " + page.getName());
        try {
            output = new FileWriter(TARGETDIR + page.getName() + ".html");
            writer = new BufferedWriter(output);
            writer.write(htmlpage.generate(page));
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

http://www.vogella.com/articles/EclipseJET/article.html

Basic concepts
A **template processor** (also known as a *template engine*, *template parser* or *template language*) is **software** or a **software component** that is designed to combine one or more templates with a **data model** to produce one or more result documents.
A (web) **template engine** is software that is designed to process web templates and content information to produce output web documents. It runs in the context of a template system.

Actual template engines @ Wikipedia

A
- Apache Velocity
- ASP.NET

C
- CakePHP
- CCAPS
- CheetahTemplate
- CodeCharge Studio
- CTPP

D
- Dylan Server Pages

E
- ERuby

F
- FreeMarker

G
- Genshi (templating language)

H
- Haml
- Handlebars (template system)

J
- JavaServer Pages
- Jinja (template engine)
- JSP Weaver

K
- Kid (templating language)

L
- Lithium (PHP framework)

M
- Mako (template engine)
- Microsoft ASP.NET Razor view engine
- Mustache (template system)

N
- Nevow

O
- Open Power Template

P
- PHPRunner

S
- Smarty

T
- Template Attribute Language
- Template Toolkit

V
- Thymeleaf
- TinyButStrong
- Toupl
- Twig (template engine)

W
- Text Template Transformation Toolkit
- VlibTemplate

X
- Web template system
- WebMacro

A web template system is composed of:

- A template engine: the primary processing element of the system;
- Content resource: any of various kinds of input data streams, such as from a relational database, XML files, LDAP directory, and other kinds of local or networked data;
- Template resource: web templates specified according to a template language;

Wikipedia does not define a general concept of template.

It does define the notion of a web template:

“A web template is a tool used to separate content from presentation in web design, and for mass-production of web documents. It is a basic component of a web template system. Web templates can be used to set up any type of website. In its simplest sense, a web template operates similarly to a form letter for use in setting up a website.”

Template and friends

- Macros
- Compile-time meta-programming
- Run-time meta-programming
- Staged computation
- Multi-stage programming
A **macro** (from the **Greek** μακρό for "big" or "far") in **computer science** is a rule or **pattern** that specifies how a certain input sequence (often a sequence of **characters**) should be mapped to a replacement input sequence (also often a sequence of characters) according to a defined procedure. The mapping process that instantiates (transforms) a macro use into a specific sequence is known as **macro expansion**. A facility for writing macros may be provided as part of a **software application** or as a part of a **programming language**.

[...]  

Macro systems that work at the level of **abstract syntax trees** are called **syntactic macros** and preserve the lexical structure of the original program.
These definitions are not very general (abstract) and not helpful to demarcate templates and related concepts. Become a Wikipedia contributor!
Relevant expressiveness

- Arguments (context) of templates.
- Text (hopefully syntax in the target language)
- Conditionals over context of template.
- Loops over context of templates.
- Declarations, statements, expressions in host language.

...
Motivation (cont’d)
Generating Java with StringTemplate for Java

StringTemplate uses ANTLR.
ANTLR uses StringTemplate.

```java
STGroup group = new STGroupFile("/tmp/test.stg");
ST st = group.getInstanceOf("decl");
st.add("type", "int");
st.add("name", "x");
st.add("value", 0);
String result = st.render(); // yields "int x = 0;"
```

;; define my-if as a macro
(define-syntax my-if
  (lambda (x)
    ;; establish that "then" and "else" are keywords
    (syntax-case x (then else)
      ()
      ;; pattern to match
      (my-if condition then yes-result else no-result)
      ;; transformer
      (syntax (if condition yes-result no-result))
    ))
)


An extended “if” with keywords for “then” and “else”.
A macro is pretty much like a template except that it is integrated into parsing/compile time of language processing and emits code (ASTs) for the same language.

Find a proper definition that fits into the context of this lecture.
This code uses preprocessor variables to deal with variability. Depending on the variables set at compile (preprocessing) time, different source-code parts are selected.


A preprocessor is pretty much like a template engine in that interpretes preprocessor statements (including invocations of preprocessor macros) in a manner that text (source code) is emitted.

Find a proper definition that fits into the context of this lecture.
XML transformation

Expression holes

XML literals as means of embedding XML in VB9.

VB9

Transformation language and data language are amalgamated on the grounds of both using XML.

XSLT


We begin to touch upon the area of language embedding (to be studied more later). In the VB9 case, VB and XML are “syntactically amalgamated.”
Questions

How aware of the target language should templates be?

What formal guarantees to provide? (Think of type safety.)

How to serve the special case of host = target language? ("macros")

How does this discuss generalize to compile-time metaprogramming?

... to run-time metaprogramming?
StringTemplate, XPand, MOF M2T, and others are not language-aware. They basically produce any text.
Non-syntactic use of C preprocessing

if (. . .)
{
  . . .
#endif

#if (. . .)
  } else {
#endif

  . . .
}

;

Grammar of the basic template language

A Formal Way from Text to Code Templates

```
tcoll = tmpl
tmpl = ≪define tn targ* ≫ tstmt* ≪enddef≫
targ = ttype tvn
tstmt = text | ≪texpr≫ | ≪expand tn texpr*≫
        | ≪if texpr≫ tstmt* ≪else≫ tstmt* ≪endif≫
        | ≪for tvn in texpr≫ tstmt* ≪endfor≫
tn  = template names
  text = text phrases
texpr = expressions
ttype = variable types
tvn = variable names
```

Collection of templates
Parameterized template definitions
Template arguments are typed.
Template statements in this order: actual text, expressions for walking context, template expansion, conditional and loop controlled by context.

Arbitrary text can be generated at this point.

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]
Static semantics of templates

Domains

\[ \tau = ttype \]
\[ T = tvn \rightarrow_{fin} \tau \]
\[ \Theta = tn \rightarrow_{fin} \tau^* \]

Principal judgements

\[ \vdash tcoll \]
\[ \Theta \vdash tmpl : \Theta \]
\[ \Theta \vdash tmpl \]
\[ \Theta, T \vdash tstmt \]
\[ T \vdash texpr : \tau \]
\[ \vdash L \tau : \tau \]
\[ \vdash_B \tau \]

(Expression types)
(Variable type table)
(Template type table)
(Well-typedness of template collections)
(Extraction of type context)
(Well-typedness of templates)
(Well-typedness of template statements)
(Well-typedness of template expressions)
(Decomposition of list types)
(Booleanness of types)

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]
Well-typedness of template collections

\[
⊥ \vdash \text{tmpl}_1 : \Theta_1 \land \cdots \land \Theta_{n-1} \vdash \text{tmpl}_n : \Theta_n \\
\land \Theta_n \vdash \text{tmpl}_1 \land \cdots \land \Theta_n \vdash \text{tmpl}_n
\]

\[\vdash \langle \text{tmpl}_1, \ldots, \text{tmpl}_n \rangle\]

Extraction of type context

\[
targ_1 = \text{ttype}_1 \ tvn_1 \land \cdots \land targ_n = \text{ttype}_n \ tvn_n \\
\land \Theta @ tn = ⊥ \land \Theta' = \Theta[tn \mapsto \langle \text{ttype}_1, \ldots, \text{ttype}_n \rangle] \\
\Theta \vdash \langle \text{define} \ tn \langle \text{targ}_1, \ldots, \text{targ} \rangle \rangle \ldots \langle \text{enddef} \rangle : \Theta'
\]

Well-typedness of templates

\[
targ_1 = \text{ttype}_1 \ tvn_1 \land \cdots \land targ_n = \text{ttype}_n \ tvn_n \\
\land T = ⊥[tvn_1 \mapsto \text{ttype}_1, \ldots, tvn_n \mapsto \text{ttype}_n] \\
\land \Theta, T \vdash \text{tstmt}_1 \land \cdots \land \Theta, T \vdash \text{tstmt}_m
\]

\[\Theta \vdash \langle \text{define} \ tn \langle \text{targ}_1, \ldots, \text{targ}_n \rangle \rangle \\
\langle \text{tstmt}_1, \ldots, \text{tstmt}_m \rangle \langle \text{enddef} \rangle\]

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]

Well-typedness of statements

\( \Theta, T \vdash text \)

\[
T \vdash texpr : \tau \\
\Theta, T \vdash \langle \text{expr} \rangle
\]

(1) \( T \vdash texpr : \tau \land \vdash_L \tau : \tau' \)

(2) \( T' = T[tvn \mapsto \tau'] \)

(3) \( \land \Theta, T' \vdash tstmt_1 \land \cdots \land \Theta, T' \vdash tstmt_n \)

\( \Theta, T \vdash \langle \text{for} \ tvn \ \text{in} \ \text{expr} \rangle \langle tstmt_1, \ldots, tstmt_n \rangle \langle \text{endfor} \rangle \)

(1) \( T \vdash texpr : \tau \land \vdash_B \tau \)

(2) \( \land \Theta, T \vdash tstmt_1 \land \cdots \land \Theta, T \vdash tstmt_m \)

\( \Theta, T \vdash \langle \text{if} \ \text{expr} \rangle \\
\langle tstmt_1, \ldots, tstmt_n \rangle \langle \text{else} \rangle \\
\langle tstmt_{n+1}, \ldots, tstmt_m \rangle \langle \text{endif} \rangle \)

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]
Dynamic semantics of templates

Domains

\[ \nu \]

\[ \beta = \text{true} \mid \text{false} \]

\[ \varphi = \text{text} \]

\[ \psi = \varphi^* \]

\[ T = \text{tvn} \rightarrow_{\text{fin}} \nu \]

\[ \Theta = \text{tn} \rightarrow_{\text{fin}} (\text{tvn}^* \times \text{tstmt}^*) \]

Principal judgements

\[ \vdash \text{tcoll} \Rightarrow \Theta \]

\[ \Theta \vdash \text{tmpl} \Rightarrow \Theta \]

\[ T, \Theta \vdash \text{tstmt} \Rightarrow \psi \]

\[ T \vdash \text{texpr} \Rightarrow \nu \]

\[ \vdash \nu \Rightarrow \varphi \]

\[ \vdash \nu \Rightarrow \nu^* \]

\[ \vdash \beta \]

(Generation of text)

(Extraction of code table)

(Execution of template statements)

(Evaluation of template expressions)

(Text conversion of values)

(Decomposition of lists)

(Boolean conversion of values)

(Variable environment)

(Expression values)

(Boolean values)

(Text phrases)

(Variable environment)

(Template code table)

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]

Code Table Extraction

\[ \bot \vdash \text{tmpl}_1 \Rightarrow \Theta_1 \land \cdots \land \Theta_{n-1} \vdash \text{tmpl}_n \Rightarrow \Theta_n \]
\[ \vdash \langle \text{tmpl}_1, \ldots, \text{tmpl}_n \rangle \Rightarrow \Theta \]

Code Table Extraction

\[ \text{targ}_1 = \text{ttype}_1 \ \text{tn}_1 \land \cdots \land \text{targ}_n = \text{ttype}_n \ \text{tn}_n \land \Theta' = \Theta[\text{tn} \mapsto \langle \langle \text{tn}_1, \ldots, \text{tn}_n \rangle, \langle \text{tstmt}_1, \ldots, \text{tstmt}_m \rangle \rangle] \]
\[ \Theta \vdash \llbracket \text{define} \ \text{tn} \langle \text{targ}_1, \ldots, \text{targ}_n \rangle \rrbracket \Rightarrow \Theta' \]
\[ \langle \text{tstmt}_1, \ldots, \text{tstmt}_m \rangle \]
\[ \llbracket \text{enddef} \rrbracket \]

Statement Evaluation

\[ T, \Theta \vdash \text{text} \Rightarrow \text{text} \]
\[ T \vdash \text{teexpr} \Rightarrow \nu \land \vdash \nu \Rightarrow \varphi \]
\[ T, \Theta \vdash \llbracket \text{teexpr} \rrbracket \Rightarrow \varphi \]

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]

\[ \vdash \text{tcoll} \Rightarrow \Theta \]

[Collection]

\[ \Theta \vdash \text{tmpl} \Rightarrow \Theta \]

[Extract]

\[ T, \Theta \vdash \text{tstmt} \Rightarrow \psi \]

[Text]

\[ T \vdash \text{expr} \Rightarrow \nu \land \vdash \nu \Rightarrow \varphi \]

[Expr]
1) $\Theta \mathrel{\upharpoonright} tn = \langle \langle tvn_1, \ldots, tvn_n \rangle, \langle stmt_1, \ldots, stmt_m \rangle \rangle$

2) $\forall T \vdash \text{expr}_1 \Rightarrow \nu_1 \land \cdots \land T \vdash \text{expr}_n \Rightarrow \nu_n$

3) $\forall T' = \bot[tvn_1 \mapsto \nu_1, \ldots, tvn_n \mapsto \nu_n]$

4) $\forall T', \Theta \vdash stmt_1 \Rightarrow \psi_1 \land \cdots \land T', \Theta \vdash stmt_n \Rightarrow \psi_n$

5) $\forall \psi = \langle \psi_1 : \ldots : \psi_m \rangle$

\[ T, \Theta \vdash \llparenthesis \text{expand} \ \tn \ \llbracket \text{expr}_1, \ldots, \text{expr}_n \rrbracket \rrparenthesis \Rightarrow \psi \]

1) $T \vdash \text{expr} \Rightarrow \nu \land \vdash_L \nu \Rightarrow \langle \nu_1, \ldots, \nu_m \rangle$

2) $\forall T_1 = T[tvn \mapsto \nu_1] \land \cdots \land T_m = T[tvn \mapsto \nu_m]$

3) $\forall T_1, \Theta \vdash stmt_1 \Rightarrow \psi_{1,1} \land \cdots \land T_1, \Theta \vdash stmt_n \Rightarrow \psi_{1,n}$

\[ \land \cdots \land T_m, \Theta \vdash stmt_1 \Rightarrow \psi_{m,1} \land \cdots \land T_m, \Theta \vdash stmt_n \Rightarrow \psi_{m,n} \]

4) $\forall \psi = \langle \psi_{1,1} : \ldots : \psi_{1,n} : \ldots : \psi_{m,1} : \ldots : \psi_{m,n} \rangle$

\[ T, \Theta \vdash \llparenthesis \text{for} \ \tn \ \text{in} \ \text{expr} \rrparenthesis \ \llbracket \text{stmt}_1, \ldots, \text{stmt}_n \rrbracket \ \llparenthesis \text{endfor} \rrparenthesis \Rightarrow \psi \]

1) $T \vdash \text{expr} \Rightarrow \nu \land \vdash_B \nu \Rightarrow \text{true}$

2) $\forall T, \Theta \vdash stmt_1 \Rightarrow \psi_1 \land \cdots \land T, \Theta \vdash stmt_n \Rightarrow \psi_n$

3) $\forall \psi = \langle \psi_1 : \ldots : \psi_n \rangle$

\[ T, \Theta \vdash \llparenthesis \text{if} \ \text{expr} \rrparenthesis \ \llbracket \text{stmt}_1, \ldots, \text{stmt}_n \rrbracket \ \llparenthesis \text{else} \rrparenthesis \ldots \ \llparenthesis \text{endif} \rrparenthesis \Rightarrow \psi \]

1) $T \vdash \text{expr} \Rightarrow \nu \land \vdash_B \nu \Rightarrow \text{false}$

2) $\forall T, \Theta \vdash stmt_1 \Rightarrow \psi_1 \land \cdots \land T, \Theta \vdash stmt_n \Rightarrow \psi_n$

3) $\forall \psi = \langle \psi_1 : \ldots : \psi_n \rangle$

\[ T, \Theta \vdash \llparenthesis \text{if} \ \text{expr} \rrparenthesis \ldots \ \llparenthesis \text{else} \rrparenthesis \ \llbracket \text{stmt}_1, \ldots, \text{stmt}_n \rrbracket \ \llparenthesis \text{endif} \rrparenthesis \Rightarrow \psi \]

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]

Language-aware template languages

Make sure that only *syntactically correct code* is produced.

Thus, templates have *result types: nonterminals* of target language’s grammar.

*Grammar transformations* come to the rescue:

amalgamate template and target syntax

by enhancing target language’s grammar to include templates.
Informal enhancement description

For each nonterminal of the target language’s grammar:

- Enable template expansion with such a result type.
- Enable conditionals with such a result type.
- Enable template definitions with such a result type.
- Enable nonterminal to generate text as statement.
- Enable the nonterminal as “type”.
Informal enhancement description (cont’d)

Some special cases:

- Handle morphemes.
- Degenerate case of normal nonterminals.
- Handle */+ symbols.
- Extract (“fold”) new nonterminal for symbol.
- Enable loops with such a result type.
Grammar adaptation for adding template support to the grammar of a language

introduce texpr ttype tvn tn
introduce targ = ttype tvn
foreach nt : N
    include nt = \{expand tn texpr* \}
    include nt = \{if texpr* \} nt \{else\} nt \{endif\}
    include tmpl = \{define [nt] tn targ* \} nt \{enddef\}
    include tstmt = nt
    include dn = [nt]
endforeach
foreach nt : M
    fold ntM = nt
    include ntM = \{texpr\}
    include tstmt = ntM
    include dn = [nt]
endforeach
foreach nt : L
    fold ntL = nt
    include ntL = \{for tvn in texpr\} nt \{endfor\}
    include tstmt = ntL
endforeach
introduce tcoll = tmpl*

Fig. 7. Generic adaptation script for the syntactical enhancement of a target language. We introduce rules for template expansion and for a conditional statement. Additionally, we introduce a rule for templates. Third, we enhance the definition of each morphem class. We fold each morphem class to a fresh nonterminal and include a rule for expression evaluation. Fourth, we enhance the definition of nonterminals occurring in a Kleene closure. We fold each of these domains to a fresh nonterminal and include a rule for iteration. Finally, we introduce template collections.

The result of the adaptation is a grammar for a template language particularly concerned with the target language. In this template language, templates are associated with a particular syntactical domain of the target language. Figure 8 gives an example. The upper part of the figure shows the grammar of a simple programming language PL: A program prog consists of a list of statements pstmt*. A statement is either a variable declaration, an assignment, a while loop, or a conditional statement. An expression pexpr is either an integer number, a character string, a variable, a sum, a difference, or a string concatenation. The lower part of Fig. 8 shows the resulting grammar for the template language TL.PL.

Static semantics. During the syntactical enhancement, two helper domains tstmt and dn are constructed. This allows us to use a generic model of the static semantics. The model differs only slightly from the model for TL⊥. Figure 9 highlights the modifications. In Θ, we keep the syntactical domain of the template in [Guido Wachsmuth: “A Formal Way from Text to Code Templates”]

Another rule from the generic grammar

Another rule from the generic grammar

Another rule from the generic grammar

Another rule from the generic grammar

Another rule from the generic grammar

Another rule from the generic grammar

Another rule from the generic grammar
An operator suite for grammar adaptation

adaptation = step*

step = introduce rrule | include rule | fold rule
       | foreach vn : yielder do step* endfor

rrule = rule | nt*

rule = nt = elem*

elem = nt | vn | kw | [vn] | elem *

yielder = N | M | L

nt    nonterminals
vn    variable names
kw    keywords

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]
Summary of the method

- Definition of generic template language.
- Syntax, static and dynamic semantics
- Amalgamation with target language’s grammar.
- Use grammar transformation systematically.
- Additional amalgamation with semantics (omitted).

[Guido Wachsmuth: “A Formal Way from Text to Code Templates”]
Multi-stage programming

Compute source code any time.
Motivation to MSP
Why Multi-stage Programming?

Reasons are purely economic:

• **Problem 1:** Abstraction mechanisms (functions, objects, classes…) have runtime cost
  – Result 1: You don’t use them
  – Result 2: That costs you programmer productivity

• Idea: Use generative programming (GPCE)

• **Problem 2:** Writing good generative programs is hard

Why program generation is hard

- First, there is the issue of hygiene (as in macros)
- More generally, what can we **statically guarantee** about what’s generated?
- Important: “Statically” = by looking at the generator?

<table>
<thead>
<tr>
<th>Approach</th>
<th>Build “f (x,y)”</th>
<th>Combine F X</th>
<th>Syntactic correctness?</th>
<th>Type correctness?</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Datatype</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Multi-stage programming (MSP)

- Provide abstraction mechanisms like: polymorphism, higher-order functions, exceptions, …
- Provide constructs that allow “generation”
- **Without** damaging the static typing discipline

<table>
<thead>
<tr>
<th>Approach</th>
<th>Build “f (x,y)”</th>
<th>Combine F X</th>
<th>Syntactic correctness?</th>
<th>Type correctness?</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Datatype</td>
<td>…</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MSP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
The abstract view

Batch Programming
Ex.: Interpreter
The abstract view

Multi-Stage Programming (MSP)
Ex.: Compiler
Staging the exponentiation function
Small example: Exponentiation

\[ x^0 = 1 \]
\[ x^{2n+1} = x \cdot x^{2n} \]
\[ x^{2n} = (x^n)^2 \]

Minimal example of a generic program.
Such a program is almost never used in practice.

Why?

\[ x = 2 \]
\[ n = 5 \]
\[ x^n = 32 \]
Small example: Exponentiation

\[ x^0 = 1 \]
\[ x^{2n+1} = x \cdot x^{2n} \]
\[ x^{2n} = (x^n)^2 \]

\[ P_2(x) = x^5 \]
\[ = x \cdot x^4 \]
\[ = x \cdot (x^2)^2 \]
\[ = x \cdot ((x^1)^2)^2 \]
\[ = x \cdot ((x \cdot x^0)^2)^2 \]
\[ = x \cdot ((x \cdot 1)^2)^2 \]

No recursion!

Just 4 ops!

A Naïve MSP method

1. Write traditional, single-stage program

2. Add staging annotations: brackets, escapes, and run
MetaOCaml

[GPC '03]

- Builds on the OCaml system
  - Very successful, widely used, statically typed language
  - Compiled (both byte-code and native code)
    - Has competitive performance (even for numerical apps)
    - Allows the collection of credible performance numbers
  - Substantial existing code base
- Realized by a source-to-source translation
  - Based on an elegant formal model
  - Has a clear underlying cost model (parse tree)
Small example: Exponentiation

Single stage:

\[
\begin{align*}
\text{let rec } & \text{exp}(n:\text{int}, x:\text{real}):\text{real} = \\
\text{if } & n = 0 \\
\text{then } & 1.0 \\
\text{else if } & \text{even } (n) \\
\text{then } & \text{sqr } (\text{exp}(n \text{ div } 2, x)) \\
\text{else } & x \times (\text{exp}(n - 1, x));
\end{align*}
\]
Small example: Exponentiation

Two stage:

let rec exp(n:int, x:<real>):<real> =

if n = 0
    then <1.0>
else if even (n)
    then <sqr ~(exp(n div 2,x))>
else <~x * ~(exp(n - 1,x))>;;
Small example: Exponentiation

To use the staged $\exp$ we simply write:

$$ ! \langle \text{fn} \ x \Rightarrow \sim(\exp(5,\langle x \rangle)) \rangle ;; $$

The result is:

$$ \text{fn} \ x \Rightarrow x*(\text{sqr}(\text{sqr}(x*1.0))) $$

$x*(((x*1)^2)^2)$
Timing in MetaOCaml

"Speedup" = \[(2.4 \times 10^{-3}) / (3.0 \times 10^{-4}) = 8 \times\]
MSP details
Multi-stage programming (MSP)

Three staging annotations:

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brackets:</td>
<td>( a = &lt;2*4&gt; )</td>
<td>( a=\langle2*4\rangle )</td>
</tr>
<tr>
<td>Escape:</td>
<td>( b = &lt;9+_a&gt; )</td>
<td>( b=\langle9+2*4\rangle )</td>
</tr>
<tr>
<td>Run:</td>
<td>( c = !b )</td>
<td>( c=17 )</td>
</tr>
</tbody>
</table>

Typing rules (simplified):

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X )</td>
<td>( T )</td>
<td>( &lt;X&gt; )</td>
<td>( &lt;T&gt; )</td>
</tr>
<tr>
<td>( X )</td>
<td>( &lt;T&gt; )</td>
<td>( \sim X )</td>
<td>( T )</td>
</tr>
<tr>
<td>( X )</td>
<td>( &lt;T&gt; )</td>
<td>( ! X )</td>
<td>( T )</td>
</tr>
</tbody>
</table>
Related techniques for “staging”

There is a number of ways for looking at MSP:

- **Statically typed macros**
  - But not limited to compile-time

- **Partial evaluation** (Mix, Schism, Similix, Tempo)
  - But programmer has full control over what is specialized
  - (in some cases, programmer has too much control)

- **High-level program generation** (SDRR, Genvoca, P++)

- **Runtime code generation** (Fabius, DyC, Dynamo, `C)

Common goal: **Alter order of evaluation to reduce cost of a computation**
Where can staging be used?

Lots of applications, including:

• Operating systems [Pu et al]
• Data base systems [Batory et al]
• Satellite software [Czarnezki et al]
• Domain specific languages [Consel et al]
• Graphics software [Mogensen]

*No shortage in applications. Rather, in methodology*
Typical speedups (MetaOCaml)

Program | Speedup T(P)/T(P₂)
---|---
Exp | 5…8 x
Interp(fib 15) | 4 x
Interp’(fib 15) | 44 x
RewriteCPS | 13 x
Chebyshev | 3 x

What kinds of programs can we stage?

Generic programs that pay an unnecessary cost for this generality.
“Delays lead to dangerous ends”

Be careful with recursion/loops...

let rec exp(n:int, x:<real>):<real> =

<if n = 0
  then 1.0
else if even (n)
  then sqr ~(exp(n div 2,x))
else ~x * ~(exp(n - 1,x))>;
Basic inlining using MSP

Instead of getting

<fun x -> x * (sqr (sqr (x))>

It would be nice if we can get

<fun x -> let y=x*x in x*(y*y)>

To do this, we need to consider the definition of sqr, (and change the staged exp function slightly)
Small example: Exponentiation

Two stage:

```
let rec exp(n:int, x:<real>):<real> =
if n = 0
   then <1.0>
else if even (n)
   then <sqr ~(exp(n div 2,x))>
else <~x * ~(exp(n - 1,x))>;
```

Can we get rid of this `sqr` operation?
In-lining using MSP

Original definition

```latex
let sqr x = x * x
```

Naïve staged version

```latex
let sqr x = <~x * ~x>
```

Note change in type:

Before: `int->int`  After: `<int>-><int>`

Must therefore change `exp`
Small example: Exponentiation

Two stage:

\[
\text{let rec } \text{exp}(n:\text{int}, \quad x:\langle\text{real}\rangle):\langle\text{real}\rangle \\
\quad = \\
\quad \text{if } n = 0 \\
\quad \text{then } \langle 1.0 \rangle \\
\quad \text{else if even (n)} \\
\quad \text{then } \text{sqr (exp}(n \text{ div } 2,x)) \\
\quad \text{else } \langle \sim x \ast \sim (\text{exp}(n - 1,x)) \rangle;
\]

Now our main function has even less green
Basic inlining using MSP

Problem: Using: \( \text{let } \text{sqr } x = \langle \sim x \ast \sim x \rangle \)

we get code explosion: We generate

\( \langle \text{fun } x \to x \ast (x \ast x) \ast (x \ast x) \rangle \)

Solution (from PE literature!)

\( \text{let } \text{sqr } x = \langle \text{let } y=\sim x \in y \ast y \rangle \)

Now we get (basically):

\( \langle \text{fun } x \to \text{let } y=x \ast x \in x \ast (y \ast y) \rangle \)
Partial Application vs. Partial Evaluation

“Curried application” is former, but NOT latter.

For example: \[ F \ x \ y \ z = x + y + z \]

Is same as: \[ F \ x \ y = \text{fun} \ z \rightarrow x + y + z \]

Now \[ F \ 1 \ 2 \] evaluates to \[ \text{fun} \ z \rightarrow 1 + 2 + z \]

but NOT: \[ \text{fun} \ z \rightarrow 3 + z. \]

Last step requires reduction under \( \lambda (\text{fun} \ z) \)

\[ F' \ x \ y = \langle \text{fun} \ z \rightarrow \sim (\text{lift} (x + y)) + z \rangle \]
Capture is a non-problem in MSP

Given the following definitions

\[
\begin{align*}
\text{let } t(y) &= \langle \text{fun } x \rightarrow x + \neg y \rangle \\
\text{let } p &= \langle \text{fun } x \rightarrow \neg (t \langle x \rangle) \rangle \ 10 \\
\end{align*}
\]

We do NOT get

\[
\begin{align*}
p &= \langle \text{fun } x \rightarrow (\text{fun } x \rightarrow x+x)10 \rangle \\
\end{align*}
\]

But rather

\[
\begin{align*}
p &= \langle \text{fun } x_1 \rightarrow (\text{fun } x_2 \rightarrow x_2+x_1)10 \rangle \\
\end{align*}
\]

With MSP, we don’t need to worry about names
Why is type safety challenging?

Type system should reject “bad” terms:

• Basics: \( \sim 5, \text{run } 5 \)

• Levels: \(<\text{let } f(x) = \sim x \text{ in } ...>\)

• Closedness: \(<\text{let } f(x) = \sim (\text{run}<x>) \text{ in } ...>\)
  \[\rightarrow <\text{let } f(x) = \sim x \text{ in } ...>\]

In the last case \( x \) is treated as having a code type, and \textsf{run} is used during “specialization”
Staging a simple interpreter
Domain Specific Languages (DSLs)

Examples

- Lex, yacc, VHDL, LaTeX, Excel, Tk/tk, etc

Benefits

- Abstract away unnecessary details
- Improve
  - Performance
  - Reliability, maintainability … = productivity

A functional MSP language can be particularly useful for implementing DSLs
Syntax in (Meta)OCaml

```
type exp =
    Int of int       | Var of string
| App of string * exp | Add of exp * exp
| Sub of exp * exp  | Mul of exp * exp
| Div of exp * exp  | Ifz of exp

type def =
    Declaration of string * string * exp

type prog = Program of def list * exp
```

Syntax in (Meta)OCaml

Represent

\[
\text{let rec } f\ x = \\
\text{if } x=0 \text{ then } 1 \text{ else } x*(f(x-1)) \\
\text{in } f\ 10
\]

As

\[
\text{let termFact = Program} \\
([\text{Declaration } ("f","x", \text{Ifz(Var "x", Int 1,Mul(Var"x",(App ("f", Sub(Var "x",Int 1)))))}], \\
\text{App ("f", Int 10)})
\]
The natural benchmark

Note that we can type in what we want to represent directly into OCaml and run it:

```ocaml
let rec f x =
  if x=0 then 1 else x*(f(x-1))
in f 10
```

This is NOT a generic solution.

But IT IS a natural benchmark for what constitutes good performance.
Interpreter

Runtime environments:

(* env : string -> int
   fenv : string -> (int -> int) *)

exception Yikes

let env0  = fun x -> raise Yikes
let fenv0 = env0
let ext env x v =
   fun y -> if x=y then v else env y
Interpreter

A safe interpreter for just expressions:

```ocaml
let rec eval3 e env fenv = match e with
    Int i -> Some i ...
| Div (e1,e2) ->
    (match (eval3 e1 env fenv,
            eval3 e2 env fenv) with
        (Some x, Some y)->
            if y=0 then None else Some (x/y)
        | _ -> None) ...
```

Interpreter

Same for programs/declarations

let rec peval3 p env fenv =

match p with

Program ([],e) -> eval3 e env fenv

|Program (Declaration (s1,s2,e1)::tl,e) ->

let rec f x =

  eval4 e1 (ext env s2 x)

  (ext fenv s1 f)

in peval3 (Program(tl,e)) env ...

env, fenv

p → P → i
Interpreter

• The interpreter is generic
  – It will evaluate any program at runtime (even ones provided by the user)

• The interpreter is simple and maintainable
  – It is essentially a specification of the semantics of the language
  – (Actually, it’s almost a denotational semantics)

• It runs 21x slower than our benchmark
  – Costly generic solutions are impractical, forcing us to resort to ad hoc solutions…
Staged Interpreter

We can traverse the program early:

```ocaml
let rec eval4 e env fenv = match e with
  Int i -> <Some i> ...
| Div (e1,e2) ->
  <(match (~(eval3 e1 env fenv),
            ~(eval3 e2 env fenv))
     with (Some x, Some y)->
         if y=0 then None else Some (x/y)
     | _ -> None) ... >
```

Staged Interpreter

Same for programs:

```ml
let rec peval4 p env fenv =
match p with
  Program ([], e) -> eval4 e env fenv
| Program (Declaration (s1, s2, e1)::tl, e) ->
  <let rec f x =
    ~(eval4 e1 (ext env s2 <x>))
    (ext fenv s1 <f>))
  in ~(peval4 (Program(tl, e)) env ... >
```

Staged Interpreter

• Staging helps
  – We get code 4x faster than the original interpreter
  – But that means it’s 5x slower than our benchmark…

• We would like to generate the following code:

  <let rec f x =
    if (x = 0) then 1 else (x*(f (x-1)))
  in (f 10)>

• But what does it actually generate?
Staged Interpreter

• Generated code:

```ocaml
<let rec f x =
    match Some x with
    Some x -> if x = 0 then Some 1 ...
    | None -> None
in match Some 10 with
    Some x -> f x
    | None -> None>
```

• What happened?
Staged Interpreter

• Generated code:

```plaintext
<let rec f x =
    match Some x with
    Some x -> if x = 0 then Some 1 ...
    | None -> None
in match Some 10 with
    Some x -> f x
    | None -> None>
```

• We are tagging and untagging…
Staged Interpreter

If there is ONE thing you want to take from this tutorial:

Avoid the temptation to find ways to post-process the code you generate. It can be arbitrarily better (in terms of performance and quality) to avoid generating bad code in the first place.
Staged Interpreter

The source of the problem:

```
let rec eval4 e env fenv = match e with

    Int i -> <Some i> ...

| Div (e1,e2) ->

    <(match (~(eval3 e1 env fenv),
            ~(eval3 e2 env fenv))

        with (Some x, Some y)->

            if y=0 then None else Some (x/y)

        | _ -> None) ... >
```
Interpreter (in CPS)

let rec eval5 e env fenv k =
  match e with
  | Int i -> k (Some i) ...
  | Div (e1,e2) ->
    eval5 e1 env fenv (fun r ->
    eval5 e2 env fenv (fun s ->
      match (r,s) with
      | (Some x, Some y) ->
        if y=0 then k None else k (Some (x/y))
      | _ -> k None)) ...

Staged Interpreter

let rec eval6 e env fenv k =
match e with
  Int i -> k (Some <i>) ...
| Div (e1,e2) ->
  eval6 e1 env fenv (fun r ->
    eval6 e2 env fenv (fun s ->
      match (r,s) with
        (Some x, Some y) ->
          <if ~y=0 then ~(k None)
          else ~(k (Some <~x / ~y>))>
        _ -> k None))

Summary of MSP

- Staging can lead to substantial speedups.
- Staging requires perhaps only minor annotations.
- Programmer can see when things happen.
- Programmer can understand why things are faster.
- Staging is closely related to templates, macros, partial evaluation, generative programming, and run-time code generation.
- Staging is based on sound and simple reasoning principles.
Further reading on MSP

Multi-stage Programming Homepage
http://www.cs.rice.edu/~taha/MSP/

The Converge programming language
http://convergepl.org/

Tiark Rompf, Martin Odersky: “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. http://dx.doi.org/10.1145/1942788.1868314
Hygienic macros

Avoid name capture of kinds.
(Only discussed in passing this time.)
#define INCI(i) {int a=0; ++i;}

```c
int main(void)
{
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

```c
int main(void)
{
    int a = 0, b = 0;
    {int a=0; ++a;};
    {int a=0; ++b;};
    printf("a is now %d, b is now %d\n", a, b);
    return 0;
}
```

```
```
The hygiene problem with Common Lisp macros

```
(defmacro my-unless (condition &body body)
  `(if (not ,condition)
      (progn
       ,@body)))

(flet ((not (x) x))
  (my-unless t
    (format t "This should not be printed!")))
```

“Redefining standard functions and operators, globally or locally, invokes undefined behavior according to ANSI Common Lisp. Such usage can be diagnosed by the implementation as erroneous.”

The hygiene problem with Common Lisp macros

```
(defmacro my-unless (condition &body body)
  `(if (user-defined-operator ,condition)
     (progn
      ,@body)))

(flet ((user-defined-operator (x) x))
  (my-unless t
    (format t "This should not be printed!")))
```

"The Common Lisp solution to this problem is to use packages. The my-unless macro can reside in its own package, where user-defined-operator is a private symbol in that package. The symbol user-defined-operator occurring in the user code will then be a different symbol, unrelated to the one used in the macro."

Solution strategies

Try to use unique names for the macro parameters.
Generate unique symbols as they are needed.
Generally disallow shadowing of names.
Generally disallow redefinitions of built-in functions.
Generally disallow all redefinitions globally.
Protect names by making them package-private.
Resolve clashes by hygienic transformation.

...
Hygienic macros in Scheme

“Meanwhile, languages such as Scheme that use hygienic macros prevent accidental capture and ensure referential transparency automatically as part of the macro expansion process. [...] For example, the following Scheme implementation of my-unless will have the desired behavior:"

```
(define-syntax my-unless
  (syntax-rules ()
    [(_ condition body)
      (if (not condition)
          body
          (void))]]

(let ([not (lambda (x) x)])
  (my-unless #t
    (displayln "This should not be printed!")))
```

Further reading on hygienic macros

E. E. Kohlbecker, M. Wand: "Macro-by-example: Deriving syntactic transformations from their specifications".
http://dx.doi.org/10.1145/41625.41632

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, Bruce Duba: "Hygienic macro expansion".
http://dx.doi.org/10.1145/319838.319859

David Herman, Mitchell Wand: "A theory of hygienic macros".
http://dl.acm.org/citation.cfm?id=1792878.1792884

Steven E. Ganz, Amr Sabry, Walid Taha: "Macros as multi-stage computations: type-safe, generative, binding macros in MacroML".
http://dx.doi.org/10.1145/507546.507646

Walid Taha, Patricia Johann: "Staged notational definitions".
http://dl.acm.org/citation.cfm?id=954186.954192
Concluding remarks

Template and macros systems are ubiquitous in programming.

Such systems clearly interact with syntax, types, and semantics.

"Safety" is provided only by some systems to some extent.

"Methodology" is still maturing.

Language engineers need to use, tame, and develop such systems.

Also, think of reverse/re-engineering systems that use templates et al.