A Formal Way from Text to Code Templates

Guido Wachsmuth

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
guwac@gk-metrik.de

Abstract. We present an approach to define template languages for generating syntactically correct code. In the first part of the paper, we define the syntax and semantics of a template language for text generation. We use Natural Semantics for expressing both the static and the dynamic semantics of the language. In the second part, we deal with template languages for code generation in a particular target language. We provide construction steps for the syntax and semantics of such languages. The approach is generic and can be applied to any target language.

1 Introduction

Code generation forms a central part of Model-driven Engineering (MDE). Template languages provide means to specify code generation. They are used in mature modelling frameworks, for example *Java Emitter Templates* [1] in the Eclipse Modeling Framework [2] or openArchitectureWare's *XPand* [3] in the Eclipse Graphical Modeling Framework [4]. With its *MOF Model to Text Transformation Language* (MOF M2T) [5], the Object Management Group proposes a standardised template language for model to text transformations. Beyond MDE, template languages are generally used in generative programming. For example, *StringTemplate* [6] is used for parser generation with ANTLR [7] as well as for generating web pages [8].

Template languages allow to specify code in concrete syntax. To generate code in different target languages, most template languages treat code simply as text. As a consequence, template languages do not provide any static judgement on syntactical correctness of templates with respect to a particular target language. In this paper, we investigate a formal method to develop a template language TL_λ for generating syntactically correct code in a given target language λ. We enhance the grammar of λ to derive a grammar for TL_λ. For this enhancement, we rely on well-defined grammar adaptation steps [9]. We use Natural Semantics [10] for expressing both the static and the dynamic semantics of TL_λ. The approach is generic and can be applied to any target language.

The remainder of the paper is structured as follows: In Sec. 2, we describe the formal foundations of our approach, i.e.grammar adaptation and Natural Semantics. In Sec. 3, we define syntax and semantics of the core concepts of template languages. In Sec. 4, we provide construction steps for template languages concerned with true code generation. The paper is concluded in Sec. 5.

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 109–123, 2009.

$$
\begin{aligned}
\mathit{adaptation} &= \mathit{step}^* \\
\mathit{step} &= \textbf{introduce}\ \mathit{rrule} \mid \textbf{include}\ \mathit{rule} \mid \textbf{fold}\ \mathit{rule} \\
&\quad \mid \textbf{foreach}\ \mathit{vn} : \mathit{yielder}\ \textbf{do}\ \mathit{step}^*\ \textbf{endfor} \\
\mathit{rrule} &= \mathit{rule} \mid \mathit{nt}^* \\
\mathit{rule} &= \mathit{nt} = \mathit{elem}^* \\
\mathit{elem} &= \mathit{nt} \mid \mathit{vn} \mid \mathit{kw} \mid \lfloor \mathit{vn} \rfloor \mid \mathit{elem}^{\ *} \\
\mathit{yielder} &= \textbf{N} \mid \textbf{M} \mid \textbf{L} \\
\mathit{nt} &\quad \text{nonterminals} \\
\mathit{vn} &\quad \text{variable names} \\
\mathit{kw} &\quad \text{keywords}
\end{aligned}
$$

Fig. 1. An operator suite for grammar adaptation

2 Preliminaries

Grammar adaptation. In this paper, we employ formal grammar transformations for the stepwise adaptation of grammars [9]. This approach offers several benefits: First, the derivation of the template language is formally defined by the application of well-defined adaptation operators. Second, we prevent accidental modifications of the target language since preservation properties of adaptation operators are well understood [9]. Third, the adaptation is generic and can be applied to any target language. Finally, existing tool support automates the derivation process [11].

We rely on a very restricted operator suite for this paper. Figure 1 shows its concrete syntax: First, we can **introduce** a new rule for a fresh nonterminal. We use the same operator to introduce nonterminals without a defining rule (e.g. for morphem classes). Second, we can **include** an additional rule for a nonterminal. If the nonterminal is yet undefined, **include** operates as **introduce**. Third, we can **fold** a phrase. This introduces a fresh nonterminal defined by the phrase. All occurrences of the phrase are replaced by the nonterminal. All these operators are purely constructive, i.e. they extend the language defined by the grammar [9]. Finally, we can execute operators **foreach** nonterminal in a set. There are three yielders of such sets: **N** yields all nonterminals except morphem classes, **M** yields all morphem classes, and **L** yields all nonterminals to which a Kleene closure is applied. Inside a **foreach** block, a variable provides access to the current nonterminal. $\lfloor \cdot \rfloor$ yields a nonterminal's name as a keyword.

Natural Semantics. In this paper, we use Natural Semantics [10] for the static and dynamic semantics of template languages. Typically, template languages are functional languages [6]. Describing functional languages by Natural Semantics is well understood. For example, Natural Semantics was used for both the static and dynamic semantics in the formal specification of *Standard ML* [12]. The general idea of a semantics definition in Natural Semantics is to provide axioms and inference rules for *judgements* over syntactical and semantical *domains*. A static semantics is typically described in terms of well-typedness judgements for

$$
\begin{array}{rl}
\mathit{tcoll} = & \mathit{tmpl}^* \\
\mathit{tmpl} = & \ll \textbf{define}\ \mathit{tn}\ \mathit{targ}^* \gg\ \mathit{tstmt}^*\ \ll \textbf{enddef} \gg \\
\mathit{targ} = & \mathit{ttype}\ \mathit{tvn} \\
\mathit{tstmt} = & \mathit{text} \mid \ll \mathit{texpr} \gg \mid \ll \textbf{expand}\ \mathit{tn}\ \mathit{texpr}^* \gg \\
& \mid \ll \textbf{if}\ \mathit{texpr} \gg\ \mathit{tstmt}^*\ \ll \textbf{else} \gg\ \mathit{tstmt}^*\ \ll \textbf{endif} \gg \\
& \mid \ll \textbf{for}\ \mathit{tvn}\ \textbf{in}\ \mathit{texpr} \gg\ \mathit{tstmt}^*\ \ll \textbf{endfor} \gg \\
\mathit{tn} & \text{template names} \\
\mathit{text} & \text{text phrases} \\
\mathit{texpr} & \text{expressions} \\
\mathit{ttype} & \text{variable types} \\
\mathit{tvn} & \text{variable names}
\end{array}
$$

Fig. 2. Syntactical domains of $TL_\perp$

the various syntactical domains of a language. A dynamic semantics is described in an operational style by judgements for the execution of language instances.

We follow some notational conventions in this paper: For semantical domains, we use standard domain constructors, namely products ("$\times$"), list types ("*"), and function domains ("$\rightharpoonup_{fin}$"). We restrict ourselves to pointwise definition of functions, i.e. these functions are only defined for a finite subset of the domain X in $f : X \rightharpoonup_{fin} Y$. For most $x \in X$ we have that $f(x) = \perp$. The entirely undefined function is denoted by "$\perp$". In inference rules, we reuse domain names as stems of meta-variables. Tuples and sequences are constructed via $\langle \cdot, \ldots, \cdot \rangle$. In order to concatenate several lists to a new one, we use the notation $\langle \cdot : \ldots : \cdot \rangle$. Update of a function f for a point x to return y is denoted by $f[x \mapsto y]$. Function application is highlighted using the notation $f\,@\,x$.

3 A Core Text Template Language

In this section, we provide formal semantics of the core concepts in template languages for text generation. For this purpose, we introduce $TL_\perp$, a functional language similar to StringTemplate, XPand, and even more to MOF M2T.

Syntax. Figure 2 shows a grammar of the template language $TL_\perp$. We briefly read its rules: A template collection *tcoll* consists of a list tmpl^* of templates. A template declares a list targ^* of arguments and a list tstmt^* of template statements. Template statements include simple text, expression evaluation, template expansion, conditional statement, and iteration.

Template languages typically comprise an expression language for navigating the source model. This sublanguage highly depends on the technological space of the source model. In the grammarware space, we need to navigate syntax trees. In the modelware space, we need to navigate object-oriented models. For this reason, we do not specify an expression sublanguage in detail. Instead, we specify several requirements for its syntax and semantics. Syntactically, we assume domains for expressions (*texpr*), types (*ttype*), and variables (*tvn*).

Domains

$\tau = ttype$	(Expression types)
$T = tvn \rightharpoonup_{fin} \tau$	(Variable type table)
$\Theta = tn \rightharpoonup_{fin} \tau^*$	(Template type table)

Principal judgements

$\vdash tcoll$	(Well-typedness of template collections)
$\Theta \vdash tmpl : \Theta$	(Extraction of type context)
$\Theta \vdash tmpl$	(Well-typedness of templates)
$\Theta, T \vdash tstmt$	(Well-typedness of template statements)
$T \vdash texpr : \tau$	(Well-typedness of template expressions)
$\vdash_L \tau : \tau$	(Decomposition of list types)
$\vdash_B \tau$	(Booleaness of types)

Fig. 3. Model of $TL_\perp$ static semantics

Static semantics. The domains involved in the static semantics and the principal judgements are shown in Fig. 3. For the expression sublanguage, we require a domain of types (τ), a domain for variable environments (T) to store variable types, and a judgement of well-typedness of expressions under a given environment. Furthermore, we require types for list-like data structures. We assume the judgment $\vdash_L \tau : \tau'$ to hold for a list type τ iff the list elements are of type τ'. The judgement $\vdash_B \tau$ is expected to hold iff τ is boolean-like, that is, elements of this type can be mapped to boolean values.

As for the core concepts of $TL_\perp$, we define a domain for type tables (Θ) to store the argument types of templates. Judgements concern the extraction of a type table for a given template collection, the overall well-typedness of a template collection, the well-typedness of a template definition for a given type table, and the well-typedness of a statement for a given type table and variable environment.

Figure 4 lists the inference rules for the judgements. Since most of them are straightforward, we do not discuss all these rules in detail. Let us focus on the well-typedness judgement for statements. $\Theta, T \vdash tstmt$ is meant to hold if the statement *tstmt* is well-typed in the context of Θ and T. The parameter Θ covers the template types whereas T corresponds to an environment mapping variables for template arguments or iterations to types.

The axiom [text] encodes the well-typedness of any text. The rule [exp] defines well-typedness for expression evaluation: An expression evaluation is well-typed if its expression has a type in the context of the current variable environment.

The premises of rule [expand] say that a template expansion is well-typed if (1) a table look-up for *tn* delivers the types of the template arguments and (2) the actual parameter types are subsumed by the formal ones.

The rule [for] defines well-typedness of iteration as follows: (1) The type τ of the iteration expression is determined under the current variable environment T. This type needs to be a list type with a corresponding type τ' for the list

Well-typedness of template collections $\boxed{\vdash tcoll}$

$$\frac{\bot \vdash tmpl_1 : \Theta_1 \ \wedge \ \cdots \ \wedge \ \Theta_{n-1} \vdash tmpl_n : \Theta_n \quad \wedge \Theta_n \vdash tmpl_1 \ \wedge \ \cdots \ \wedge \ \Theta_n \vdash tmpl_n}{\vdash \langle\, tmpl_1, \ldots, tmpl_n \,\rangle} \ [\textsf{collection}]$$

Extraction of type context $\boxed{\Theta \vdash tmpl : \Theta}$

$$\frac{targ_1 = ttype_1\ tvn_1 \ \wedge \ \cdots \ \wedge \ targ_n = ttype_n\ tvn_n \quad \wedge \Theta \mathbin{@} tn = \bot \ \wedge \ \Theta' = \Theta[tn \mapsto \langle\, ttype_1, \ldots, ttype_n \,\rangle]}{\Theta \vdash \ll \textbf{define}\ tn\ \langle\, targ_1, \ldots, targ \,\rangle \gg \ \ldots \ \ll \textbf{enddef} \gg : \Theta'} \ [\textsf{extract}]$$

Well-typedness of templates $\boxed{\Theta \vdash tmpl}$

$$\frac{\begin{array}{l} targ_1 = ttype_1\ tvn_1 \ \wedge \ \cdots \ \wedge \ targ_n = ttype_n\ tvn_n \\ \wedge\, T = \bot[tvn_1 \mapsto ttype_1, \ldots, tvn_n \mapsto ttype_n] \\ \wedge\, \Theta, T \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T \vdash tstmt_m \end{array}}{\begin{array}{l} \Theta \vdash \ll \textbf{define}\ tn\ \langle\, targ_1, \ldots, targ_n \,\rangle \gg \\ \quad \langle\, tstmt_1, \ldots, tstmt_m \,\rangle \\ \quad \ll \textbf{enddef} \gg \end{array}} \ [\textsf{template}]$$

Well-typedness of statements $\boxed{\Theta, T \vdash tstmt}$

$$\Theta, T \vdash text \ [\textsf{text}]$$

$$\frac{T \vdash texpr : \tau}{\Theta, T \vdash \ll texpr \gg} \ [\textsf{expr}]$$

$$\frac{(1) \ \ \Theta \mathbin{@} tn = \langle\, \tau_1, \ldots, \tau_n \,\rangle \quad (2) \wedge T \vdash texpr_1 : \tau_1 \ \wedge \ \cdots \ \wedge \ T \vdash texpr_n : \tau_n}{\Theta, T \vdash \ll \textbf{expand}\ tn\ \langle\, texpr_1, \ldots, texpr_n \,\rangle \gg} \ [\textsf{expand}]$$

$$\frac{\begin{array}{l} (1) \ \ T \vdash texpr : \tau \ \wedge \ \ \vdash_L \tau : \tau' \\ (2) \wedge \tau' = T[tvn \mapsto \tau'] \\ (3) \wedge \Theta, T' \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T' \vdash tstmt_n \end{array}}{\Theta, T \vdash \ll \textbf{for}\ tvn\ \textbf{in}\ texpr \gg \ \langle\, tstmt_1, \ldots, tstmt_n \,\rangle \ \ll \textbf{endfor} \gg} \ [\textsf{for}]$$

$$\frac{(1) \ \ T \vdash texpr : \tau \ \wedge \ \ \vdash_B \tau \quad (2) \wedge \Theta, T \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T \vdash tstmt_m}{\begin{array}{l} \Theta, T \vdash \ll \textbf{if}\ texpr \gg \\ \quad \langle\, tstmt_1, \ldots, tstmt_n \,\rangle \\ \quad \ll \textbf{else} \gg \\ \quad \langle\, tstmt_{n+1}, \ldots, tstmt_m \,\rangle \\ \quad \ll \textbf{endif} \gg \end{array}} \ [\textsf{if}]$$

Fig. 4. Rules of $TL_\bot$ static semantics

Domains

ν	(Expression values)
$\beta = \textbf{true} \mid \textbf{false}$	(Boolean values)
$\varphi = \mathit{text}$	(Text phrases)
$\psi = \varphi^*$	
$T = \mathit{tvn} \rightharpoonup_{\mathit{fin}} \nu$	(Variable environment)
$\Theta = \mathit{tn} \rightharpoonup_{\mathit{fin}} (\mathit{tvn}^* \times \mathit{tstmt}^*)$	(Template code table)

Principal judgements

$\vdash \mathit{tcoll} \Rightarrow \Theta$	(Extraction of code table)
$\Theta \vdash \mathit{tmpl} \Rightarrow \Theta$	
$T, \Theta \vdash \mathit{tstmt} \Rightarrow \psi$	(Execution of template statements)
$T \vdash \mathit{texpr} \Rightarrow \nu$	(Evaluation of template expressions)
$\vdash \nu \Rightarrow \varphi$	(Text conversion of values)
$\vdash_L \nu \Rightarrow \nu^*$	(Decomposition of lists)
$\vdash_B \nu \Rightarrow \beta$	(Boolean conversion of values)

Fig. 5. Model of $TL_\perp$ dynamic semantics

elements. (2) A new variable environment T' is retrieved by updating the type of the iteration variable *tvn* to τ'. (3) All statements in the iteration need to be well-typed under the new environment.

The rule [if] for conditional statements reads similarly: (1) The type τ of the condition expression is determined. This type needs to be boolean-like. (2) All statements in a conditional statement need to be well-typed in the current context.

Dynamic semantics. While the static semantics is concerned with well-typedness judgements, the dynamic semantics provides judgements about text generation. Figure 5 shows the model of the dynamic semantics. We use a style that emphasises the similarities of the models of static and dynamic semantics. While Θ covers the *template types* in the static case, it models the *template code* in the dynamic case. Similar variations apply to the principal judgements. That is, code table extraction corresponds to type table extraction, text generation out of template statements to well-typedness of template statements, and text generation out of template expressions to type assignment for template expressions.

For the expression sublanguage, we require a domain of values (ν), a domain for variable environments (T) to store variable values, and a judgement for evaluating expressions under a given environment. The structure of this judgement implies side-condition free evaluation. This ensures model view separation [13]. Furthermore, we assume the judgment $\vdash_L \nu \Rightarrow \langle \nu_1, \ldots, \nu_n \rangle$ to hold for a list-like data structure ν iff $\nu_1, \ldots, \nu_n$ are the elements of this structure. The judgement $\vdash_B \nu \Rightarrow \beta$ is expected to hold iff ν can be converted into the boolean value β. Finally, we require a judgement $\vdash \nu \Rightarrow \varphi$ for the conversion of values to text.

Figure 6 lists the inference rules for the judgements. Again, we do not discuss all these rules in detail. Let us focus on the text generation judgement for

Well-typedness of template collections $\boxed{\vdash tcoll}$

$$\frac{\bot \vdash tmpl_1 : \Theta_1 \ \wedge \ \cdots \ \wedge \ \Theta_{n-1} \vdash tmpl_n : \Theta_n \quad \wedge \Theta_n \vdash tmpl_1 \ \wedge \ \cdots \ \wedge \ \Theta_n \vdash tmpl_n}{\vdash \langle\, tmpl_1, \ldots, tmpl_n \,\rangle} \quad [\text{collection}]$$

Extraction of type context $\boxed{\Theta \vdash tmpl : \Theta}$

$$\frac{targ_1 = ttype_1\ tvn_1 \ \wedge \ \cdots \ \wedge \ targ_n = ttype_n\ tvn_n \quad \wedge\, \Theta \mathbin{@} tn = \bot \ \wedge \ \Theta' = \Theta[tn \mapsto \langle\, ttype_1, \ldots, ttype_n \,\rangle]}{\Theta \vdash \ll \textbf{define}\ tn\ \langle\, targ_1, \ldots, targ \,\rangle \gg \ \ldots \ \ll \textbf{enddef} \gg : \Theta'} \quad [\text{extract}]$$

Well-typedness of templates $\boxed{\Theta \vdash tmpl}$

$$\frac{targ_1 = ttype_1\ tvn_1 \ \wedge \ \cdots \ \wedge \ targ_n = ttype_n\ tvn_n \quad \wedge\, T = \bot[tvn_1 \mapsto ttype_1, \ldots, tvn_n \mapsto ttype_n] \quad \wedge\, \Theta, T \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T \vdash tstmt_m}{\begin{array}{l}\Theta \vdash \ll \textbf{define}\ tn\ \langle\, targ_1, \ldots, targ_n \,\rangle \gg \\ \quad \langle\, tstmt_1, \ldots, tstmt_m \,\rangle \\ \quad \ll \textbf{enddef} \gg\end{array}} \quad [\text{template}]$$

Well-typedness of statements $\boxed{\Theta, T \vdash tstmt}$

$$\Theta, T \vdash text \quad [\text{text}]$$

$$\frac{T \vdash texpr : \tau}{\Theta, T \vdash \ll texpr \gg} \quad [\text{expr}]$$

$$\frac{(1) \ \ \Theta \mathbin{@} tn = \langle\, \tau_1, \ldots, \tau_n \,\rangle \quad (2) \wedge T \vdash texpr_1 : \tau_1 \ \wedge \ \cdots \ \wedge \ T \vdash texpr_n : \tau_n}{\Theta, T \vdash \ll \textbf{expand}\ tn\ \langle\, texpr_1, \ldots, texpr_n \,\rangle \gg} \quad [\text{expand}]$$

$$\frac{\begin{array}{l}(1) \ \ T \vdash texpr : \tau \ \wedge \ \ \vdash_L \tau : \tau' \\ (2) \wedge T' = T[tvn \mapsto \tau'] \\ (3) \wedge \Theta, T' \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T' \vdash tstmt_n\end{array}}{\Theta, T \vdash \ll \textbf{for}\ tvn\ \textbf{in}\ texpr \gg \ \langle\, tstmt_1, \ldots, tstmt_n \,\rangle \ \ll \textbf{endfor} \gg} \quad [\text{for}]$$

$$\frac{(1) \ \ T \vdash texpr : \tau \ \wedge \ \ \vdash_B \tau \quad (2) \wedge \Theta, T \vdash tstmt_1 \ \wedge \ \cdots \ \wedge \ \Theta, T \vdash tstmt_m}{\begin{array}{l}\Theta, T \vdash \ll \textbf{if}\ texpr \gg \\ \quad \langle\, tstmt_1, \ldots, tstmt_n \,\rangle \\ \quad \ll \textbf{else} \gg \\ \quad \langle\, tstmt_{n+1}, \ldots, tstmt_m \,\rangle \\ \quad \ll \textbf{endif} \gg\end{array}} \quad [\text{if}]$$

Fig. 6. Rules of $TL_\perp$ dynamic semantics

statements. $T, \Theta \vdash tstmt \Rightarrow \psi$ is meant to hold if the statement *tstmt* generates a list of text phrases ψ in the context of Θ and T.

The axiom [text] states that text generates itself. The rule [expr] defines text generation for template expression. The expression is evaluated in the context of the current variable environment to a value which in turn is converted to text.

The premises of rule [expand] specifies text generation by template expansion as follows: (1) A table look-up for *tn* delivers the parameter names and the statements of the template. (2) Parameter expressions are evaluated under the current variable environment T and (3) the results are assigned to the parameter names in a fresh variable environment T'. (4) The statements are evaluated under the new variable environment and the resulting text phrases are concatenated.

The rule [for] defines iterative text generation as follows: (1) The value ν of the iteration expression is determined under the current variable environment T. This value needs to be a list of elements $\nu_1, \ldots, \nu_m$. (2) New variable environments $T_1, \ldots, T_m$ are retrieved by updating the value of the iteration variable *tvn* to the corresponding value. (3) All statements in the iteration are executed under each new environment and (4) the generated text phrases are concatenated.

The rules [then] and [else] for conditional statements read similarly: (1) The value ν of the condition expression is evaluated under the current variable environment T and converted into a boolean value. (2) For **true**, the rule [then] executes the first branch. For **false**, the rule [else] executes the second branch. All statements in the branch are executed and (3) the generated text phrases are concatenated.

4 Generating Code Template Languages

In this section, we provide a generic approach to derive a template language from a target language. The resulting template language ensures the generation of syntactically correct code with respect to the target language. The derivation is split into three phases: First, we rely on grammar adaptation to syntactically enhance the target language with constructs for template definitions. This steps results in a grammar for the template language. Second, we rely on Natural Semantics to define the static semantics of the resulting template language. This includes a static judgement about the syntactical correctness of templates with respect to the target language. Finally, we rely again on Natural Semantics to define the dynamic semantics of the resulting template language.

Syntactical enhancement. We use grammar adaptation to enhance the grammar of a target language. Figure 7 gives the corresponding adaptation script which can be applied generically to any target language. First, we introduce expressions, types, variable names, template names, and template arguments. While names are typically morphem classes, expressions and types need to refer to the grammar of an expression language. Second, we enhance the definition of each nonterminal except morphem classes. We include rules for template expansion and for a conditional statement. Additionally, we include a rule for templates. Third, we enhance the definition of each morphem class. We fold each

```
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⌋
endfor
foreach nt : M
    fold nt_M = nt
    include nt_M = ≪texpr≫
    include tstmt = nt_M
    include dn = ⌊nt⌋
endfor
foreach nt : L
    fold nt_L = nt
    include nt_L = ≪for tvn in texpr≫ nt ≪endfor≫
    include tstmt = nt_L
endfor
introduce tcoll = tmpl*
```

Fig. 7. Generic adaptation script for the syntactical enhancement of a target language

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 $TL_{\perp}$. Figure 9 highlights the modifications. In Θ, we keep the syntactical domain of the template in addition to the parameter types. In the well-typedness judgement for template statements, we assign the addressed syntactical domain. Furthermore, we add a judgement yielding the syntactical domain of an expression when converted into text.

$$
\begin{array}{rl}
\mathit{pprog} = & \textbf{begin } \mathit{pstmt}^* \textbf{ end} \\
\mathit{pstmt} = & \mathit{pvn} : \mathit{ptype} \mid \mathit{pvn} := \mathit{pexpr} \mid \textbf{while } \mathit{pexpr} \textbf{ do } \mathit{pstmt}^* \textbf{ od} \\
& \mid \textbf{if } \mathit{pexpr} \textbf{ then } \mathit{pstmt}^* \textbf{ else } \mathit{pstmt}^* \textbf{ fi} \\
\mathit{pexpr} = & \mathit{pvn} \mid \mathit{in} \mid \mathit{cs} \mid \mathit{pexpr} + \mathit{pexpr} \mid \mathit{pexpr} - \mathit{pexpr} \mid \mathit{pexpr} \parallel \mathit{pexpr} \\
\mathit{ptype} = & \textbf{int} \mid \textbf{string} \\
\mathit{pvn} & \text{variable names} \\
\mathit{in} & \text{integer numbers} \\
\mathit{cs} & \text{character strings}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{tcoll} = & \mathit{tmpl}^* \\
\mathit{tmpl} = & \ll\textbf{define pprog } \mathit{tn}\ \mathit{targ}^* \gg \mathit{pprog}\ \ll\textbf{enddef}\gg \\
& \mid \ll\textbf{define pstmt } \mathit{tn}\ \mathit{targ}^* \gg \mathit{pstmt}\ \ll\textbf{enddef}\gg \\
& \mid \ll\textbf{define pexpr } \mathit{tn}\ \mathit{targ}^* \gg \mathit{pexpr}\ \ll\textbf{enddef}\gg \\
& \mid \ll\textbf{define ptype } \mathit{tn}\ \mathit{targ}^* \gg \mathit{ptype}\ \ll\textbf{enddef}\gg \\
\mathit{targ} = & \mathit{ttype}\ \mathit{tvn} \\
\mathit{pprog} = & \textbf{begin } \mathit{pstmt}^*_L \textbf{ end} \\
& \mid \ll\textbf{expand } \mathit{tn}\ \mathit{texpr}^* \gg \\
& \mid \ll\textbf{if } \mathit{texpr}^* \gg\ \mathit{pprog}\ \ll\textbf{else}\gg\ \mathit{pprog}\ \ll\textbf{endif}\gg \\
\mathit{pstmt} = & \mathit{pvn}_M : \mathit{ptype} \mid \mathit{pvn}_M := \mathit{pexpr} \mid \textbf{while } \mathit{pexpr} \textbf{ do } \mathit{pstmt}^*_L \textbf{ od} \\
& \mid \textbf{if } \mathit{pexpr} \textbf{ then } \mathit{pstmt}^*_L \textbf{ else } \mathit{pstmt}^*_L \textbf{ fi} \\
& \mid \ll\textbf{expand } \mathit{tn}\ \mathit{texpr}^* \gg \\
& \mid \ll\textbf{if } \mathit{texpr}^* \gg\ \mathit{pstmt}\ \ll\textbf{else}\gg\ \mathit{pstmt}\ \ll\textbf{endif}\gg \\
\mathit{pstmt}_L = & \mathit{pstmt} \mid \ll\textbf{for } \mathit{tvn} \textbf{ in } \mathit{texpr}\gg\ \mathit{pstmt}\ \ll\textbf{endfor}\gg \\
\mathit{pvn}_M = & \mathit{pvn} \mid \ll \mathit{texpr} \gg \\
& \ldots \\
\mathit{tstmt} = & \mathit{pprog} \mid \mathit{pstmt} \mid \mathit{pexpr} \mid \mathit{ptype} \mid \mathit{pvn}_M \mid \mathit{in}_M \mid \mathit{cs}_M \mid \mathit{pstmt}_L \\
\mathit{dn} = & \textbf{pstmt} \mid \textbf{pexpr} \mid \textbf{ptype} \mid \textbf{pvn} \mid \textbf{in} \mid \textbf{cs}
\end{array}
$$

Fig. 8. Syntactical domains of a simple programming language PL and its corresponding template language TL_{PL}

Domains

$\Theta = tn \rightharpoonup_{fin} (\tau^* \times \boxed{dn})$ (Template types and domains)

Principal judgements

$\Theta, T \vdash tstmt : \boxed{dn}$ (Well-typedness of template statements)

$\boxed{\vdash \tau : dn}$ (Syntactical domains of expression types)

Fig. 9. Generic model of TL_{PL} static semantics (excerpt)

There are two kinds of inference rules. First, we provide generic rules for language constructs introduced during the syntactical enhancement. Second, we need to generate inference rules covering original constructs of the target language. The generic rules deal with template statements introduced by the syntactical enhancement. Only minor modifications are needed to the inference rules of the static semantics of $TL_\perp$. These modifications deal with domain

assignment for templates, statements, and expressions. The upper part of Fig. 10 presents the affected rules.

The lower part of the figure shows some of the inference rules generated for TL_{PL}. For each morphem class m, we generate an axiom of the form

$$\Theta, T \vdash m : \lfloor m \rfloor$$

where $\lfloor m \rfloor$ yields the name of m. For each grammar rule $nt = \langle e_1, \ldots, e_n \rangle$, we generate an inference rule of the form

$$\frac{premise_1 \quad \wedge \quad \cdots \quad \wedge \quad premise_m}{\Theta, T \vdash \langle p_1, \ldots, p_n \rangle : \lfloor nt \rfloor}$$

Each right-hand side element e_i is mapped to a corresponding pattern p_i in the inference rule and premises might be added.

1. A nonterminal nt is mapped to a fresh variable v of the corresponding domain. The premise $\Theta, T \vdash v : \lfloor nt \rfloor$ is added.
2. A Kleene closure nt^* is mapped to a list pattern $\langle v_1, \ldots, v_k \rangle$ with fresh variables. The premises $\Theta, T \vdash v_1 : \lfloor nt \rfloor \quad \wedge \quad \ldots \quad \wedge \quad \Theta, T \vdash v_k : \lfloor nt \rfloor$ are added.
3. A morphem class m is mapped to a fresh variable v of the corresponding domain for m_M which resulted from folding m during the syntactical enhancement. The premise $\Theta, T \vdash v : \lfloor m \rfloor$ is added.
4. A keyword is mapped to itself. No premise is added.

Dynamic semantics. As for the static semantics, we can reuse the model of $TL_{\perp}$ dynamic semantics. The only modification affects the code table: We only need to store single template statements instead of statement lists. Again, there are two kinds of inference rules, generated and generic ones. The upper part of Fig. 11 shows the generic rules. Modifications to inference rules of $TL_{\perp}$ dynamic semantics are highlighted.

The lower part of the figure shows generated rules for TL_{PL}. Generation is quite similar to the static case. For each morphem class m, we generate an axiom of the form

$$\Theta, T \vdash m \Rightarrow m$$

stating that a morphem generates itself. For each grammar rule $nt = \langle e_1, \ldots, e_n \rangle$, we generate an inference rule of the form

$$\frac{premise_1 \quad \wedge \quad \cdots \quad \wedge \quad premise_m}{\Theta, T \vdash \langle p_1, \ldots, p_n \rangle \Rightarrow \langle \psi_1 : \ldots : \psi_n \rangle}$$

The mapping of right-hand side element e_i to a corresponding pattern p_i is the same as in the static case. The following premises are added:

1. For a nonterminal, the premise $\Theta, T \vdash v \Rightarrow \psi_i$ is added.
2. For a Kleene closure, premises $\Theta, T \vdash v_1 \Rightarrow \psi_{i,1} \quad \wedge \quad \ldots \quad \wedge \quad \Theta, T \vdash v_1 \Rightarrow \psi_{i,m} \quad \wedge \quad \psi_i = \langle \psi_{i,1} : \ldots : \psi_{i,k} \rangle$ are added.
3. For a morphem class, the premise $\Theta, T \vdash v \Rightarrow \psi_i$ is added.
4. For a keyword kw, the premise $\psi_i = kw$ is added.

Extraction of type context $\boxed{\Theta \vdash tmpl : \Theta}$

$$\frac{\begin{array}{l} targ_1 = ttype_1\ tvn_1 \ \wedge\ \cdots\ \wedge\ targ_n = ttype_n\ tvn_n \\ \wedge\ \Theta' = \Theta[tn \mapsto \langle\, \langle\, ttype_1, \ldots, ttype_n \,\rangle, \boxed{dn}\,\rangle] \end{array}}{\Theta \vdash \ll\!\textbf{define}\ \boxed{dn}\ tn\ \langle\, targ_1, \ldots, targ_n \,\rangle \gg\ \ldots\ \ll\!\textbf{enddef}\gg : \Theta'}\ [\textsf{extract}]$$

Well-typedness of templates $\boxed{\Theta \vdash tmpl : dn}$

$$\frac{\begin{array}{l} targ_1 = ttype_1\ tvn_1 \ \wedge\ \cdots\ \wedge\ targ_n = ttype_n\ tvn_n \\ \wedge\ T = \bot[tvn_1 \mapsto ttype_1, \ldots, tvn_n \mapsto ttype_n] \\ \wedge\ \Theta, T \vdash tstmt : \boxed{dn} \end{array}}{\Theta \vdash \ll\!\textbf{define}\ \boxed{dn}\ tn\ \langle\, targ_1, \ldots, targ_n \,\rangle \gg\ \boxed{tstmt}\ \ll\!\textbf{enddef}\gg : \boxed{dn}}\ [\textsf{template}]$$

Well-typedness of statements $\boxed{\Theta, T \vdash tstmt : dn}$

$$\frac{T \vdash texpr : \tau \ \wedge\ \boxed{\vdash \tau : dn}}{\Theta, T \vdash \ll texpr \gg : \boxed{dn}}\ [\textsf{expr}]$$

$$\frac{\begin{array}{l} \Theta\,@\,tn = \langle\, \langle\, \tau_1, \ldots, \tau_n \,\rangle, \boxed{dn}\,\rangle \\ \wedge\ T \vdash texpr_1 : \tau_1 \ \wedge\ \cdots\ \wedge\ T \vdash texpr_n : \tau_n \end{array}}{\Theta, T \vdash \ll\!\textbf{expand}\ tn\ \langle\, texpr_1, \ldots, texpr_n \,\rangle\ \gg : \boxed{dn}}\ [\textsf{expand}]$$

$$\frac{\begin{array}{l} T \vdash texpr : \tau \ \wedge\ \vdash_L \tau : \tau' \\ \wedge\ T' = T[tvn \mapsto \tau'] \\ \wedge\ \Theta, T' \vdash tstmt : \boxed{dn} \end{array}}{\Theta, T \vdash \ll\!\textbf{for}\ tvn\ \textbf{in}\ texpr \gg\ \boxed{tstmt}\ \ll\!\textbf{endfor}\gg : \boxed{dn}}\ [\textsf{for}]$$

$$\frac{\begin{array}{l} T \vdash texpr : \tau \ \wedge\ \vdash_B \tau \\ \wedge\ \Theta, T \vdash tstmt_1 : \boxed{dn} \ \wedge\ \Theta, T \vdash tstmt_2 : \boxed{dn} \end{array}}{\Theta, T \vdash \ll\!\textbf{if}\ texpr \gg\ \boxed{tstmt_1}\ \ll\!\textbf{else}\gg\ \boxed{tstmt_2}\ \ll\!\textbf{endif}\gg : \boxed{dn}}\ [\textsf{if}]$$

$$\frac{(2)\ \ \Theta, T \vdash pstmt_1 : \textbf{pstmt} \ \wedge\ \cdots\ \wedge\ \Theta, T \vdash pstmt_n : \textbf{pstmt}}{\Theta, T \vdash \textbf{begin}\ \langle\, pstmt_1, \ldots, pstmt_n \,\rangle\ \textbf{end} : \textbf{pprog}}\ [\textsf{pprog}]$$

$$\frac{\begin{array}{l} (3)\ \ \Theta, T \vdash pvn_M : \textbf{pvn} \\ (1) \wedge \Theta, T \vdash ptype : \textbf{ptype} \end{array}}{\Theta, T \vdash pvn_M : ptype : \textbf{pstmt}}\ [\textsf{pstmt}_1]$$

$$\vdots$$

$$\Theta, T \vdash cs : \textbf{cs}\ [\textsf{cs}]$$

Fig. 10. Rules of TL_{PL} static semantics (excerpt)

Code table extraction $\boxed{\boxed{\Theta \vdash tmpl \Rightarrow \Theta}}$

$$\frac{\begin{array}{l} targ_1 = ttype_1\ tvn_1 \ \wedge \ \cdots \ \wedge \ targ_n = ttype_n\ tvn_n \\ \wedge\ \Theta' = \Theta[tn \mapsto \langle\, \langle\, tvn_1, \ldots, tvn_n \,\rangle, \boxed{tstmt}\,\rangle] \end{array}}{\Theta \ \vdash \ \ll\mathbf{define}\ \boxed{dn}\ tn\ \langle\, targ_1, \ldots, targ_n \,\rangle \gg \boxed{tstmt}\ \ll\mathbf{enddef}\gg \Rightarrow \Theta'}\ [\mathsf{extract}]$$

Statement evaluation $\boxed{\boxed{T, \Theta \vdash tstmt \Rightarrow \psi}}$

$$\frac{\begin{array}{l} \Theta\,@\,tn = \langle\, \langle\, tvn_1, \ldots, tvn_n \,\rangle, \boxed{tstmt}\,\rangle \\ \wedge\ T \ \vdash texpr_1 \Rightarrow \nu_1 \ \wedge \ \cdots \ \wedge \ T \ \vdash texpr_n \Rightarrow \nu_n \\ \wedge\ T' = \bot[tvn_1 \mapsto \nu_1, \ldots, tvn_n \mapsto \nu_n] \\ \wedge\ T', \Theta \ \vdash \boxed{tstmt} \Rightarrow \boxed{\psi} \end{array}}{T, \Theta \ \vdash \ll\mathbf{expand}\ tn\ \langle\, texpr_1, \ldots, texpr_n \,\rangle \gg\Rightarrow \boxed{\psi}}\ [\mathsf{expand}]$$

$$\frac{\begin{array}{l} T \ \vdash texpr \Rightarrow \nu \ \wedge \ \vdash_L \nu \Rightarrow \langle\, \nu_1, \ldots, \nu_m \,\rangle \\ \wedge\ T_1 = T[tvn \mapsto \nu_1] \ \wedge \ \cdots \ \wedge \ T_m = T[tvn \mapsto \nu_m] \\ \wedge\ T_1, \Theta \ \vdash \boxed{tstmt} \Rightarrow \boxed{\psi_1} \\ \wedge\ \cdots \\ \wedge\ T_m, \Theta \ \vdash \boxed{tstmt} \Rightarrow \boxed{\psi_m} \\ \wedge\ \psi = \langle\, \boxed{\psi_1 : \ldots : \psi_m} \,\rangle \end{array}}{T, \Theta \ \vdash \ll\mathbf{for}\ tvn\ \mathbf{in}\ texpr \gg \ \boxed{tstmt}\ \ll\mathbf{endfor}\gg\Rightarrow \psi}\ [\mathsf{for}]$$

$$\frac{\begin{array}{l} T \ \vdash texpr \Rightarrow \nu \ \wedge \ \vdash_B \nu \Rightarrow \mathbf{true} \\ \wedge\ T, \Theta \ \vdash \boxed{tstmt} \Rightarrow \boxed{\psi} \end{array}}{T, \Theta \ \vdash \ll\mathbf{if}\ texpr \gg \ \boxed{tstmt}\ \ll\mathbf{else}\gg \ldots \ll\mathbf{endif}\gg\Rightarrow \psi}\ [\mathsf{then}]$$

$$\frac{\begin{array}{l} T \ \vdash texpr \Rightarrow \nu \ \wedge \ \vdash_B \nu \Rightarrow \mathbf{false} \\ \wedge\ T, \Theta \ \vdash \boxed{tstmt} \Rightarrow \boxed{\psi} \end{array}}{T, \Theta \ \vdash \ll\mathbf{if}\ texpr \gg \ldots \ll\mathbf{else}\gg \boxed{tstmt}\ \ll\mathbf{endif}\gg\Rightarrow \psi}\ [\mathsf{else}]$$

$$\frac{\begin{array}{l} (4)\ \ \psi_1 = \mathbf{begin} \\ (2)\wedge \Theta, T \ \vdash pstmt_1 \Rightarrow \psi_{2,1} \ \wedge \ \cdots \ \wedge \ \Theta, T \ \vdash pstmt_n \Rightarrow \psi_{2,n} \\ \wedge\ \psi_2 = \langle\, \psi_{2,1} : \ldots : \psi_{2,n} \,\rangle \\ (4)\wedge \psi_3 = \mathbf{end} \end{array}}{\Theta, T \ \vdash \mathbf{begin}\ \langle\, pstmt_1, \ldots, pstmt_n \,\rangle\ \mathbf{end} \Rightarrow \langle\, \psi_1 : \psi_2 : \psi_3 \,\rangle}\ [\mathsf{pprog}]$$

$$\frac{\begin{array}{l} (3)\ \ \Theta, T \ \vdash pvn_M \Rightarrow \psi_1 \\ (4)\wedge \psi_2 = : \\ (1)\wedge \Theta, T \ \vdash ptype \Rightarrow \psi_3 \end{array}}{\Theta, T \ \vdash pvn_M : ptype \Rightarrow \langle\, \psi_1 : \psi_2 : \psi_3 \,\rangle}\ [\mathsf{pstmt_1}]$$

$$\vdots$$

Fig. 11. Rules of TL_{PL} dynamic semantics (excerpt)

5 Conclusion

Contribution. We give formal semantics to a core template language for text generation. This way, we provide a starting point for semantics definitions of template languages like MOF M2T. Furthermore, we make a transition from text to true code generation. We show how a template language concerned with a particular target language can be derived from the target language itself. The resulting template language has clear semantics and ensures the syntactically correctness of generated code. This contributes to the E in MDE. The approach is generic and can be applied to any target language. In general, this paper contributes to software language engineering.

Related Work. In the technological space of grammarware, some program transformation languages like ASF [14] and Stratego/XT [15] allow to specify program transformations in the concrete syntax of the object language. This enables code generation based on concrete syntax. [16], provides a case study for code generation with Stratego/XT. Furthermore, the benefits of using concrete syntax in transformations and of a judgement about the syntactical correctness of transformations are discussed. In [17], a generic method to integrate target language grammars into arbitrary program transformation languages is presented. The method is based on modular syntax definitions in the syntax definition formalism SDF. In contrast to this approach, we prevent manual integration by using formal grammar adaptation steps. Furthermore, we address the semantics of the transformation language. In general, our approach is related to the embedding of languages, e.g. SQL, into host languages [18,19], e.g. Java [20].

Future Work. In this paper, we concern syntactical correctness of generated code. In a next step, the static semantics of the target language should be taken into account. This includes a restricted form of well-typedness checks (in terms of the target language) for templates. Another important point is tool support. When it comes to the target language, current template editors miss many useful features like error highlighting and code completion. This inhibits productive template engineering. Our approach is a starting point to overcome these shortcomings: Grammars and Natural Semantics allow for generic prototypical tool support. Thus, we can directly develop language tools on base of the formal syntax and semantics of a derived template language.

Acknowledgement. This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK). The author is indebted to Ralf Lämmel for providing him with layout templates for the Natural Semantics descriptions.

References

1. The Eclipse Foundation: Java Emitter Templates (JET) (2008), http://www.eclipse.org/modeling/emf/
2. The Eclipse Foundation: Eclipse Modeling Framework (EMF) (2007), http://www.eclipse.org/modeling/emf/
3. OpenArchitectureWare: XPand (2008), http://www.openarchitectureware.org
4. The Eclipse Foundation: Eclipse Graphical Modeling Framework (GMF) (2008), http://www.eclipse.org/gmf/
5. Object Management Group: MOF Model to Text Transformation Language, version 1.0 (January 2008)
6. Parr, T.J.: A functional language for generating structured text. Draft (2006)
7. Parr, T.J., Quong, R.W.: Antlr: a predicated-ll(k) parser generator. Softw. Pract. Exper. 25(7), 789–810 (1995)
8. Parr, T.J.: Intelligent web site page generation (2007)
9. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
10. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
11. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. ENTCS 44(2) (2001)
12. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
13. Parr, T.J.: Enforcing strict model-view separation in template engines. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) WWW 2004, pp. 224–233. ACM, New York (2004)
14. van den Brand, M., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, p. 365. Springer, Heidelberg (2001)
15. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.16: components for transformation systems. In: Hatcliff, J., Tip, F. (eds.) PEPM 2006, pp. 95–99. ACM, New York (2006)
16. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 183–198. Springer, Heidelberg (2008)
17. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
18. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. In: Consel, C., Lawall, J.L. (eds.) GPCE 2007, pp. 3–12. ACM, New York (2007)
19. Gao, J., Heimdahl, M., Van Wyk, E.: Flexible and extensible notations for modeling languages. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 102–116. Springer, Heidelberg (2007)
20. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)