Бреслав А. А.

**Автоматизация разработки механизмов композиции в предметно-ориентированных языках**

# Оглавление

# Глава 1.
# Предварительные сведения из области технологии разработки компьютерных языков

В этой главе...

## § 1. Постановка задачи

Целью настоящей работы является **автоматизация разработки механизмов композиции в предметно-ориентированных языках**. Для достижения поставленной цели необходимо решить следующие задачи:

- Разработать автоматизированный метод расширения предметно-ориентированных языков поддержкой типизированных макроопределений.
- Разработать автоматизированный метод расширения предметно-ориентированных языков поддержкой аспектов.
- Формально описать разработанные методы и обосновать их корректность.
- Продемонстрировать применение разработанных методов, применив их к предметно-ориентированному языку, полезному для решения практических задач.

# Глава 2.
# Расширение предметно-ориентированных языков механизмами композиции

## § 1. Автоматическое построение языков, поддерживающих типизированные макроопределения

В настоящем разделе описан метод, позволяющий по описанию языка построить описание более богатого языка, поддерживающего композицию с помощью типизированных макроопределений. Мы будем называть такой пополненный язык *макро-языком*, построенным на базе исходного языка.

**Замечание 2.1** (О терминологии)**.** *В англоязычной литературе используется термин* macro *[24,* **?***, 74]. В качестве русского перевода этого термина употребляется либо слово "макрос", являющееся в сущности сленговым и полученное транслитерацией множественного числа "macros", либо слово "макроопределение", соответствующее более узкому по смыслу термину "macro definition". Близкой по смыслу альтернативой является термин "шаблон" (англ. "template"), используемый в языке* C++ *[77] и ряде других [***?***, 26, 13].*

*Мы считаем термин "macro" более подходящим для наших целей, и будем, следуя традиции, использовать более формальный вариант его перевода —* макроопределение.

### 1.1. Неформальное описание механизма композиции, основанного на макроопределениях

Наиболее известными системами, использующими макроопределения, являются язык программирования LISP [75] и препроцессор языка C [44, **?**]. В обоих случаях макроопределения служат для обогащения языка новыми конструкциями, которые преобразуются в базовые конструкции языка во время компиляции[1], этот процесс называется *разворачиванием* макроопределений.

Приведем пример использования макроопределений в языке C: пусть нам требуется реализовать односвязные списки. Для этого в первую очередь необходимо описать *структуру* элемента списка, например, для списка целых чисел эта структура будет выглядеть следующим образом:

```
struct IntList {
    struct IntList* next;
    int data;
};
```

Для списка элементов другого типа, например, строк, структура будет очень похожей, но тип поля `data` будет отличаться:

```
struct StrList {
    struct StrList* next;
    char* data;
};
```

Дублировать описания для каждого нового типа элементов не очень удобно, поэтому мы напишем макроопределение, которое по данному типу элемента генерирует описание соответствующей структуры[2]:

```
#define DEFLIST(name, type) struct name { \
    struct name *next; \
    type data; \
};
```

После директивы `#define` следует *имя* макроопределения, а за ним в скобках — *параметры*. Весь остальной текст — это *тело* макроопределения (знак “\” используется для подавления перевода строки).

[1] Понятие “время компиляции” в данном контексте является собирательным и противопоставляется понятию “время выполнения программы”.

[2] Задача, которую мы решаем в этом примере с помощью макроопределений языка C, более эффективно решается в языке C++ с помощью *шаблонных структур*, которые можно рассматривать как узкоспециализированную разновидность макроопределений.

Для того, чтобы использовать данное макроопределение, достаточно написать его имя и передать в скобках значения параметров (то есть *аргументы*), например:

```
DEFLIST(StrList, char*)
```

В результате *разворачивания* данного определения аргументы будут подставлены в тело вместо соответствующих параметров и мы получим определение структуры `StrList`, приводившееся выше. Аналогично можно получить определения структуры `IntList`, а также структуры элемента списка для любого типа. Заметим, что разворачивание происходит во время компиляции и результатом является исходный текст на языке C, который, в свою очередь, транслируется в машинный код, причем транслятор ничего не знает о том, были использованы макроопределения или нет.

Обобщая сведения, приведенные в данном примере, можно предложить следующий список свойств, присущих механизму макроопределений[3]:

- Макроопределение состоит из *списка параметров* и *тела* и имеет уникальное *имя*.
- Тело макроопределения содержит конструкции на целевом языке (в нашем примере — на языке C) и ссылки на параметры. Также тело может содержать обращения к другим макроопределениям.
- При разворачивании ссылки на параметры в теле макроопределения заменяются значениями соответствующих аргументов.
- Разворачивание происходит во время компиляции программы.

Макроопределения представляют собой достаточно гибкий механизм повторного использования. В принципе, этот механизм не зависит от целевого языка, в который разворачиваются макроопределения.

[3]Такое обобщение имеет смысл, поскольку в различных языках и системах макроопределения функционируют схожим образом.

В частности, в языке C поддержка макроопределений обеспечивается препроцессором CPP — независимым программным средством, обрабатывающим исходный код *до* начала работы собственно компилятора языка C. Препроцессор CPP может работать с любым текстом, не только с исходным кодом на языке C, следовательно он (или аналогичный механизм) может применяться для повторного использования и в других языках, в частности в предметно-ориентированных, делая их более пригодными для использования в промышленных проектах.

Однако чисто текстовый препроцессор обладает одним важным недостатком: корректность результата разворачивания макроопределений никак не гарантируется, поскольку препроцессор манипулирует простым текстом и “не знает” о синтаксисе целевого языка.

Вернемся к примеру макроопределения, приведенному выше. Если программист допустит ошибку при использовании макроопределения `DEFLIST`, а именно перепутает порядок аргументов, что случается не так уж редко, препроцессор послушно выполнит свою работу:

```
DEFLIST(char*, StrList)
```

первратится в

```
struct char* {  // error: expected '{' before 'char'
    struct char* *next;
    StrList data;
};
```

Получившийся код синтаксически некорректен, и компилятор, получив этот текст на вход, выдаст сообщение об ошибке:

```
DEFLIST(char*, StrList) // error: expected '{' before 'char'
```

В итоге ошибка программиста обнаружена, но сообщение, выданное компилятором, записано в терминах программы, полученной после разворачивания макроопределений, и совсем не помогает программисту исправить ситуацию. Чтобы разобраться, в чем проблема, придется вручную рассмотреть код, полученный на выходе препроцессора, что является существенным затруднением при разработке больших проектов. Описанная здесь проблема является основной причиной, по

которой профессиональные программисты зачастую избегают широкого использования возможностей макроопределений в программах на C [**?**].

Итак, чисто текстовый препроцессор позволяет легко обеспечить поддержку макроопределений в любом языке, но не обеспечивает своевременного обнаружения ошибок, что затрудняет разработку. В данной главе мы рассмотрим метод реализации макроопределений, который также пригоден для любого языка, но обеспечивает контроль корректности аргументов макроопределений с помощью специальной системы типов, что позволяет избежать проблем, присущих чисто текстовым препроцессорам.

!!!!!!!!!!!!!!!!!!!!!!!!!!!! Данный подход получил обобщение в виде концепции *многоуровневых языков* (multi-stage languages), реализациями которой являются MACROML [24] и TEMPLATE HASKELL [**?**]. !!!!!!!!!!!!!!!!!!!!!!!!!!!!

## 1.2. Базовый язык макроопределений

Все языки шаблонов имеют довольно большую общую часть, называемую *базовым языком шаблонов*. Мы детально опишем эту часть в данном разделе. Детали, специфичные для конкретных языков шаблонов, будут описаны в следующих разделах.

Целевая мета-модель базового языка шаблонов (Рис. 1.2) описывает основные понятия, на которых этот язык строится. Поскольку эта же мета-модель будет использована для описания аспектов, мы не используем в ней слова “шаблон” (template), “параметр” (parameter) и т.д., но приводим следующее “отображение” терминологии мета-модели на терминологию языка шаблонов:

- Шаблон представляется классом `Abstraction` и характеризуется именем, параметрами и телом.
- Параметру соответствует класс `Variable`.

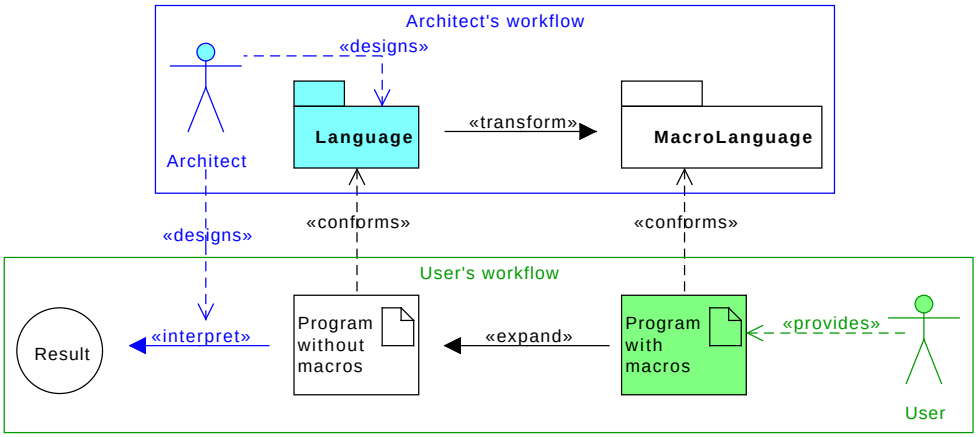


Рис. 1.1: Создание и использование языка с поддержкой макроопределений

- Шаблонные выражения представляются абстрактным классом `Term`, имеющим в базовом языке только два конкретных подкласса:
  - использование шаблонного параметра (`VariableRef`);
  - применение шаблона (`Application`), описываемое ссылкой на шаблон и аргументами, подставляемыми вместо формальных параметров.

Данная мета-модель также использует типы, указывать которые не обязательно. Тип характеризуется уникальным именем и кратностью, соответствующей одиночному вхождению или одной из операций “`?`”, “$\star$” или “$+$”.

Контекстно-свободная грамматика базового языка шаблонов приведена в Лист. 1.1 (в нотации GRAMMATIC). Эта грамматика сама является шаблоном, поскольку базовый язык необходимо расширять для того, чтобы построить язык шаблонов на основе некоторого конкретного языка. В данном описании имеется два параметра, позволяющих добавлять новые виды выражений (`domainSpecificTerms`) и новые типы (`domainSpecificTypes`). Поскольку GRAMMATIC исполь-

```
class Type {
  attr typeName : String;
  attr multiplicity
            : Multiplicity;
}

enum Multiplicity {
  VAL, MANY_VAL, REF
}

class Abstraction {
  attr name : String;
  val parameters : Variable*;
  val body : Term;
  val type : Type?;
}

abstract class Term {}

class VariableRef extends Term {
  ref variable : Variable;
}

class Application extends Term {
  ref abstraction : Abstraction;
  val arguments : Term*;
}

class Variable {
  attr name : String;
  val type : Type?;
}
```

Рис. 1.2: Базовая мета-модель языка шаблонов

зует данный базовый язык шаблонов, мы не приводим здесь примеры использования его синтаксиса.

```
template Templates<domainSpecificTypes : Expression*,
        domainSpecificTerms : Expression*> : Grammar {
    abstraction
        : 'template' NAME <List variable, ','> type?
            '{' term '}';
    variable : NAME type?;
    type : ':' typeName ('?' | '*' | '+')?;
    typeName
        : basicType
        : <?domainSpecificTypes> ;
    basicType : 'Integer' | 'String' | 'Boolean' | 'Character' ;
    term
        : genericTerm
        : <?domainSpecificTerms> ;
    genericTerm
        : templateVariableRef
        : '<' NAME term* '>' ;
}
```

Листинг 1.1: Базовый синтаксис языка шаблонов

Семантика языка шаблонов задается операцией *разворачивания*, описанной в композиционном стиле правилами на Рис. 1.3. Мы обозначаем результат разворачивания шаблонов в выражении $e$ как $\mathcal{I}_\gamma[e]$, где $\gamma$ (“*среда*”) является множеством значений параметров шаблонов

вида $p = e$, где $p$ — параметр, а $e$ — шаблонное выражение. Как видно из правила *app-inst*, когда разворачивается применение шаблона, среда пополняется информацией о текущих значениях параметров, при этом вызов происходит по значению, то есть аргументы разворачиваются до обработки тела вызываемого шаблона.

$$\frac{\{p = e\} \subseteq \gamma}{\mathcal{I}_\gamma\,[<?p>] = e}\ \text{var-inst}$$

$$\frac{\mathbf{template}\,(T <p_1, \ldots, p_n> \{b\})}{\gamma' = \bigcup_{i=1}^{n} \{p_i = \mathcal{I}_\gamma\,[a_i]\} \qquad \mathcal{I}_\gamma\,[<T\ a_1, \ldots, a_n>] = \mathcal{I}_{\gamma \cup \gamma'}\,[b]}\ \text{app-inst}$$

Рис. 1.3: Базовая семантика языка шаблонов

Для того, чтобы гарантировать, что результат применения шаблона будет корректным с точки зрения целевой мета-модели, мы определяем систему типов для языка шаблонов. Базовые правила этой системы типов приведены на Рис. 1.4, они должны быть дополнены специфичными правилами для поддержки конкретного языка. Отношение $\preceq$ является специфичным для расширяемого языка и не определяется в базовой системе типов, а лишь используется.

В правилах *list* и *set* использовано отношение $Item(x, \tau)$, означающее, что $x$ может быть элементом коллекции $\tau$. Определение этого отношения дано на том же рисунке и сводится к тому, что внутри коллекции типа $\tau$ можно использовать не только одиночные значения этого типа, но и переменные, сами являющиеся коллекциями, что соответствует возможности вставки сразу нескольких элементов.

Дополнительно к правилам типизации, на язык шаблонов накладывается следующее требование: *Никакой шаблон не должен быть достижим из своего тела по ссылкам*. Это требование гарантирует отсутствие рекурсии в определениях шаблонов.

## 1.3. Генерация языка шаблонов

$$\frac{}{\Gamma \cup \{v : \tau\} \vdash <?v> : \tau}\ \text{var}$$

$$\frac{\Gamma \cup \bigcup_{i=1}^{n} \{p_i : \tau_i\} \vdash b : \sigma}{\Gamma \vdash \mathbf{template}\,(T<p_1 : \tau_1, \ldots, p_n : \tau_n> : \sigma\,\{b\})}\ \text{abstr}$$

$$\frac{\Gamma \vdash \mathbf{template}\,(T<p_1 : \tau_1, \ldots, p_n : \tau_n> : \sigma\,\{b\}) \quad \forall i : [1:n].\ \Gamma \vdash a_i : \tau_i}{\Gamma \vdash <T\ a_1, \ldots, a_n> : \sigma}\ \text{appl}$$

$$\frac{}{\Gamma \vdash NULL : \tau^{?}}\ \text{null} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : \tau^{?}}\ \text{relax} \qquad \frac{\Gamma \vdash x : \tau^{+}}{\Gamma \vdash x : \tau^{*}}\ \text{relax}^{+}$$

$$\frac{\Gamma \vdash x : \tau \quad \tau \preceq \sigma}{\Gamma \vdash x : \sigma}\ \text{subtype}$$

$$\frac{}{\Gamma \vdash [] : \tau*}\ \text{elist} \qquad \frac{\forall i \in [1:n].\ \Gamma \vdash Item(t_i, \tau)}{\Gamma \vdash [t_1, \ldots, t_n] : \tau^{+}}\ \text{list}$$

$$\frac{}{\Gamma \vdash \{\} : \tau*}\ \text{eset} \qquad \frac{\forall i \in [1:n].\ \Gamma \vdash Item(t_i, \tau)}{\Gamma \vdash \{t_1, \ldots, t_n\} : \tau^{+}}\ \text{set}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash Item(t, \tau)}\ \text{item} \qquad \frac{\Gamma \vdash v : \tau^{*}}{\Gamma \vdash Item(<?v>, \tau)}\ \text{item*} \qquad \frac{\Gamma \vdash v : \tau^{+}}{\Gamma \vdash Item(<?v>, \tau)}\ \text{item}^{+}$$

Рис. 1.4: Базовая система типов языка шаблонов

Для того, чтобы построить язык шаблонов на основе заданного языка $L$, необходимо расширить базовую мета-модель, алгоритм разворачивания и систему типов конструкциями и правилами, специфичными для этого языка.

Пусть целевая мета-модель языка $L$ обозначается $\mathfrak{M}$, тогда мета-модель соответствующего языка шаблонов, $\mathcal{TM}$, состоит из классов, полученных применением преобразования $\mathcal{TC}\,(\bullet)$, описанного на Рис. 1.5, к каждому классу мета-модели $\mathfrak{M}$. Эта мета-модель использует как классы базовой мета-модели шаблонов (`Term`), так и примитивные типы и перечисления мета-модели $\mathfrak{M}$.

Преобразование $\mathcal{TC}\,(\bullet)$ сопоставляет каждому классу соответствующий тип шаблонного выражения. При этом ссылки перенаправляются на класс `Term`, поскольку вместо конкретного объекта может выступать шаблонное выражение.

Аналогичное преобразование конкретного синтаксиса требует

$$\mathcal{TC}\left(\textbf{class}\ \langle S, R, A\rangle\right) = \textbf{class}\ \langle \texttt{Term}, \mathcal{TR}\left(R\right), \mathcal{TA}\left(A\right)\rangle$$
$$\mathcal{TC}\left(C^{*}\right) = \mathcal{TC}\left(C\right)^{*}$$
$$\mathcal{TC}\left(C^{+}\right) = \mathcal{TC}\left(C\right)^{+}$$
$$\mathcal{TC}\left(C^{?}\right) = \mathcal{TC}\left(C\right)^{?}$$
$$\mathcal{TR}\left(\textbf{ref}\langle r : T\rangle\right) = \textbf{ref}\langle r : \texttt{Term}\rangle$$
$$\mathcal{TR}\left(\textbf{var}\langle r : T\rangle\right) = \textbf{val}\langle r : \texttt{Term}\rangle$$
$$\mathcal{TA}\left(\textbf{attr}\langle a : T\rangle\right) = \textbf{attr}\langle a : T\rangle$$

Рис. 1.5: Преобразование конструкций языка в шаблонные выражения

пополнения грамматики из Лист. 1.1 конструкциями языка $L$. Кроме того, сами эти конструкции должны допускать использование шаблонных выражений. Таким образом сначала грамматика языка $L$ преобразуется следующим образом: к каждому нетерминалу $N$, соответствующему классу в целевой мета-модели, добавляется продукция $N \rightarrow$ `term`, где `term` — это нетермниал для шаблонных выражений из грамматики, приведенной в Лист. 1.1. Это преобразование можно выразить следующим аспектным правилом в языке GRAMMATIC:

```
#{class} : <p : Production+>;
    instead p : (p (: term));
```

Теперь необходимо придать значения параметрам грамматики базового языка шаблонов. Параметр `domainSpecificTypes` замещается множеством литералов, содержащих имена классов и примитивных типов из целевой мета-модели $L$, объединенных с помощью операции альтернативы, например

```
'Sequence' | 'Alternative' | 'Literal'
```

Параметр `domainSpecificTerms` замещается множеством всех нетерминалов преобразованной грамматики языка $L$.

Построенная таким образом грамматика может оказаться неоднозначной. К сожалению, задача обнаружения неоднозначности является алгоритмически неразрешимой [**?**], и автоматизированные средства могут применять лишь эвристические методы для ее решения. В настоящий момент обнаружение и устранение неоднозначностей воз-

лагается на разработчика.

Для обеспечения функционирования специфичных конструкций процедура разворачивания шаблонов пополняется правилами следующего вида:

$$\frac{t = \mathcal{TC}\,(C)\,@id\,\{r_i = rv_i,\ a_j = av_j\}}{\mathcal{I}_\gamma\,[t] = \mathcal{TC}\,(C)\,@id'\,\{r_i = \mathcal{I}_\gamma\,[rv_i]\,,\ a_j = av_j\}}\ \text{ds-inst(C)}$$

Задача таких правил — развернуть шаблонные выражения, находящиеся внутри специфичных конструкций, поэтому все эти правила однотипны и просто осуществляют рекурсивные вызовы на значениях всех ссылок, выходящих из данного объекта. Поскольку в графе объектов могут быть циклы, в процессе разворачивания поддерживается служебное множество уже обработанных объектов, что соответствует стандартному алгоритму обхода графа в глубину [**?**].

Согласно приведенному выше правилу, результатом разворачивания шаблонного выражения является другое шаблонное выражение, полученное разворачиванием всех ссылок на параметры и применений шаблонов. Будем называть такие выражения имеющими *нормальную форму*. Для того, чтобы получить из выражения $e$ в нормальной форме экземпляр мета-модели $\mathfrak{M}$, необходимо его преобразовать. Такое преобразование (обозначаемое $\mathfrak{M}\,(e)$) весьма просто: для каждого объекта класса $\mathcal{TC}\,(C)$ создается объект класса $C$ с той же структурой, то есть ссылки трансформируются рекурсивно, а атрибуты копируются. Ниже мы будем подразумевать выполнение данного преобразования после разворачивания шаблонов.

Система типов также пополняется правилами для специфичных конструкций. Эти правила имеют следующий вид:

$$\frac{\begin{array}{ll} x = \mathcal{TC}\,(C)\,@id\,\{r_i = v_i; a_j = v'_j\} & r_i = \mathcal{TR}\,(\rho_i : \tau_i) \\ \mathcal{TM} \Vdash x : \mathcal{TC}\,(C) & \Gamma \vdash v_i : \tau_i \end{array}}{\Gamma \vdash x : C}\ \text{ds-type(C)}$$

Однотипность этих правил также объясняется их рекурсивной природой: они нужны только для того, чтобы проверить типы в шаблонных выражениях внутри данного объекта, если он сам удовлетворяет требованиям мета-модели $\mathcal{TM}$.

Заметим, что шаблонное выражение, являющееся объектом класса $\mathcal{TC}(C)$ типизируется самим классом $C$. Это необходимо для соблюдения требований к наследованию. Поскольку в мета-модели $\mathcal{TM}$ все классы наследуются от класса `Term`, отношения наследования в ней не соответствуют таким же отношениям в $\mathfrak{M}$. Поэтому в правиле *subtype* на Рис. 1.4 отношение $\preceq$ задается мета-моделью $\mathfrak{M}$, а не $\mathcal{TM}$, и типы тоже берутся из $\mathfrak{M}$.

Чтобы показать, что ограничения, накладываемые системой типов адекватны требованиям мета-модели, докажем следующую лемму.

**Лемма 2.1** (О нормальных формах)**.** *Если шаблонное выражение* $e$ *имеет нормальную форму и* $\vdash e : \tau$*, то* $\Psi, \mathfrak{M} \Vdash \mathfrak{M}(e) : \tau$.

*Доказательство.* Достаточно заметить, что в дереве вывода для $\vdash e : \tau$ правила *var*, *abstr*, *appl*, *item** и *item*$^+$ не встречаются, а остальные правила в системе типов для шаблонов имеют прямые аналоги в системе требований мета-модели на Рис. **??**. □

## 1.4. Структурная корректность

Система типов накладывает ограничения на шаблоны. Шаблоны и шаблонные выражения, удовлетворяющие правилам типизации, при разворачивании порождают конечные объекты, удовлетворяющие требованиям целевой мета-модели. Для того, чтобы убедиться в этом, покажем, что для системы типов и семанитки языка шаблонов выполняются свойства *сохранения типов* и *нормализации* [65].

Как отмечалось выше, необходимым условием корректного поведения является тот факт, что функция разворачивания получает на

вход выражение, в котором соблюдаются правила системы типов. Это условие формализовано в следующем определении.

**Определение 2.1.** Среда $\gamma$ называется *согласованной с контекстом* $\Gamma$, если все ее элементы имеют допустимые типы:

$$\forall p \,:\, \{p = e\} \subseteq \gamma \Rightarrow \begin{cases} \{p : \tau\} \subseteq \Gamma \\ \Gamma \vdash e : \tau \end{cases}$$

Теперь докажем первое из упомянутых выше свойств.

**Теорема 2.2** (О сохранении типов)**.** *Если среда* $\gamma$ *согласована с контекстом* $\Gamma$ *и* $\Gamma \vdash e : \tau$*, то* $\Gamma \vdash \mathcal{I}_\gamma [e] : \tau$*. Другими словами, преобразование* $\mathcal{I}[\bullet]$ *сохраняет типы.*

*Доказательство.* Будем вести индукцию по определению $\mathcal{I}_\gamma [e]$.
**База.** Правило *ds-inst(C)* сохраняет типы, поскольку из объекта класса $\mathcal{TC}(C)$ получается объект того же класса, а правило *ds-type(C)* выводит тип из класса объекта.
**Переход.** В правиле *app-inst* среда расширяется, и нам необходимо показать, что результат согласован с контекстом. Это обеспечивается требованиями к типам параметров, накладываемыми правилом *appl* и способом формирования контекста в правиле *abstr*. Из этого по предположению индукции следует, что правило *app-inst* сохраняет типы.

Так же по предположению правило *var-inst* сохраняет типы. □

Свойство нормализации формулируется следующим образом:

**Теорема 2.3** (О нормализации)**.** *Если среда* $\gamma = \cup\{p_i = e_i\}$ *согласована с контекстом* $\Gamma$*, все* $e_i$ *имеют нормальную форму и* $\Gamma \vdash e : \tau$*, то результат вычисления* $\mathcal{I}_\gamma [e]$ *имеет нормальную форму.*

*Доказательство.* Будем вести индукцию по определению $\mathcal{I}_\gamma [[]\, \bullet]$.
**База.** Результат применения правила *var-inst* имеет нормальную форму, поскольку все элементы среды имеют нормальную форму. **Переход.** В правиле *app-inst* среда пополняется значениями, имеющими нормальную форму, поэтому по предположению индукции вызовы

$\mathcal{I}_\gamma\left[a_i\right]$ и $\mathcal{I}_{\gamma\cup\gamma'}\left[b\right]$ заканчиваются за конечное число шагов и результат имеет нормальную форму.

Правило *ds-inst(C)* заменяет значения ссылок результатами разворачивания, имеющими нормальную форму, следовательно и результат применения правила имеет нормальную форму. □

Требование о том, чтобы все элементы среды имели нормальную форму выполняется для пустой среды, следовательно теорема применима для разворачивания шаблонных выражений, применяемых на практике.

Мы показали, что функция $\mathcal{I}\left[\bullet\right]$ корректно разворачивает все шаблонные параметры и применения шаблонов, а также что она не нарушает структурной корректности с точки зрения мета-модели. Результатом применения этой функции всегда является константное шаблонное выражение, которое, как отмечалось выше, тривиальным образом преобразуется в экземпляр мета-модели $\mathfrak{M}$. Таким образом, описанный здесь механизм шаблонов работает корректно.

## 1.5. Вывод типов

Как отмечалось выше, в большинстве случаев типы в объявлениях шаблонов можно опускать, поскольку они могут быть реконструированы автоматически. Для этого применяется алгоритм, аналогичный использованному в GRAMMATIC $^{SDT}$ (см. Главу **??** и [65]). Тип переменной выводится исходя из двух соображений: (а) какие аргументы ей присваиваются в применениях шаблона и (б) в каком контексте она используется в теле шаблона, то есть, если на использование переменной указывает ссылка $\mathcal{TR}\left(R\right)$, то учитывается тип ссылки $R$. Если система ограничений, построенная таким образом, не имеет решения, генерируется сообщение об ошибке типизации. Такой подход позволяет для переменной найти наиболее широкий тип объектов, которые могут быть подставлены на ее место.

Возникающие в процессе реконструкции неоднозначности разрешаются следующим образом: если параметр используется непосредственно внутри коллекции типа $T$, он получает тип $T^*$ и позволяет добавить в эту коллекцию ноль или более элементов. Исключение составляет случай, когда параметр является единственным элементом коллекции, в которой мета-модель требует наличия хотя бы одного элемента: в этом случае параметр получает тип $T^+$.

## § 2. Автоматическое построение языков, поддерживающих аспекты

В настоящем разделе описан метод расширения языков механизмом композиции на основе аспектов. Данный метод базируется на методе построения языков шаблонов, описанном выше.

### 2.1. Базовый язык аспектов

Основные понятия языка аспектов двойственны понятиям языка шаблонов. Шаблонное выражение соответствует образцу (используется класс `Term`), но, в отличие от случая шаблонов, вместо того, чтобы подставлять значения вместо переменных, требуется решать обратную задачу: найти значения переменных, при которых результат разворачивания данного шаблона будет структурно эквивалентен данному объекту.

Напомним, что в образцах переменные связываются с выражениями. Так в GRAMMATIC можно записать `'x' ?e=(?a='z' ';')`, и переменной `e` будут сопоставляться последовательности вида "`'z' ';'`", при этом символ `'z'` будет сопоставлен переменной `a`. При использовании шаблонных выражений это достигается за счет использования применений шаблонов. Данный пример соответствует выражению `<T1 <T2 'z'>>`, где шаблоны `T1` и `T2` определены следующим образом:

```
template T1 <e> { 'x' <?e> }
template T2 <a> { <?a> ';'}
```

За счет такой аналогии, для аспектов удается использовать тот же способ расширения мета-модели, что и в случае шаблонов, а также соответствующую систему типов. Дополнительно в базовую мета-модель вводятся классы для аспекта и аспектного правила, сопоставляющего одному срезу набор советов (см. Рис. 2.1). Кроме того, базовый набор шаблонных выражений пополняется классом `Wildcard`, представляющим подстановочные знаки, которым нет аналога в языке шаблонов.

```
class Aspect {
    rules : AspectRule*;
}
class AspectRule {
    pointcut : Term;
    advice : Map<Variable, Term>;
}
class Wildcard extends Term {
    type : Type;
}
```

Листинг 2.1: Классы базовой мета-модели языка аспектов

Конкретный синтаксис строится следующим образом: к синтаксису шаблонов добавляются конструкции для описания срезов (нетерминальный символ `aspectTerm`):

```
aspectTerm
    : genericAspectTerm
    : <?domainSpecificTerm> ;
genericAspectTerm
    : aspectVariable type? '=' aspectTerm // Определение переменной
    : aspectVariable                      // Ссылка на переменную
    : '<' '?' type '>' ;                  // Подстановочный знак
aspectVariable : '?' NAME;
```

- Определение переменной имеет форму `?имя : Тип = выражение`, где указание типа можно опустить.
- Ссылка на переменную имеет форму `?имя`.
- Подстановочный знак имеет форму `<? : Тип>`.

Кроме того, добавляются конструкции для описания аспектных правил:

```
aspect : aspectRule*;
aspectRule : 'on' aspectTerm 'perform' substRule* ;
substRule : 'instead' NAME ':' term ;
```

"Операторные скобки" `on...perform` введены для того, чтобы избежать синтаксической неоднозначности, возникновение которой весьма вероятно в данном случае. Это, безусловно, не гарантирует отсутствия неоднозначности, поэтому, как и в случае шаблонов, конкретные синтаксические конструкции могут требовать ручной доработки. Смысл аспектного правила таков: если образец между `on` и `perform` успешно сопоставляется с объектом в модели, то все советы `instead` применяются следующим образом: значения переменной в левой части заменяются результатом разворачивания шаблонного выражения в правой. Шаблонное выражение может содержать ссылки на переменные, связанные в образце.

Дополнительная генерация специализированных конструкций в случае аспектов не требуется: они уже сгенерированы при добавлении шаблонов.

## 2.2. Семантика сопоставления с образцом

Семантика языка аспектов описывается двумя алгоритмами: сопоставления с образцом описано в данном подразделе, а применение аспектного правила — в следующем.

### 2.2.1. Мульти-среды

Понятие среды, введенное при формализации шаблонов, необходимо расширить для случая аспектов, поскольку в этом случае одной переменной может быть сопоставлено несколько значений, которые являются *структурно идентичными* (обозначается $\cong$), то есть либо совпадают, либо являются точными копиями друг друга. Такие структуры будем называть *мульти-средами* и обозначать $\Upsilon$.

Основная операция на мульти-средах — извлечение элемента, она возвращает (возможно, пустое) множество элементов, сопоставленных данной переменной: $\Upsilon(var) = \{e_1, \ldots, e_n\}$. Для построения мульти-сред будем использовать два конструктора и две операции композиции. Простейшая мульти-среда — пустая — обозначается конструктором $\{\}$, при этом

$$\forall v: \ \{\}(v) = \emptyset.$$

Мульти-среда, сопоставляющая переменной $v$ одно значение $e$ обозначается конструктором $\{v \mapsto e\}$, при этом

$$\begin{cases} \{v \mapsto e\}(v) = \{e\} \\ \{v \mapsto e\}(x) = \emptyset, \quad \text{при } x \neq v \end{cases}.$$

Более сложные мульти-среды строятся с помощью операций *объединения* $\uplus$ и *замены* $\Cap$, заданных следующими правилами:

$$(\Upsilon_1 \uplus \Upsilon_2)(v) = \Upsilon_1(v) \cup \Upsilon_2(v)$$

$$(\Upsilon_1 \Cap \Upsilon_2)(v) = \begin{cases} \Upsilon_2(v), & \text{если } \Upsilon_2(v) \neq \bot \\ \Upsilon_1(v), & \text{если } \Upsilon_2(v) = \bot \end{cases}$$

Специальный элемент $\bot$ не является мульти-средой, но операции композиции доопределяются на нем следующим образом:

$$\begin{aligned} \Upsilon \ \uplus \ \bot \ &= \ \bot \\ \bot \ \uplus \ \Upsilon \ &= \ \bot \\ \Upsilon \ \Cap \ \bot \ &= \ \bot \\ \bot \ \Cap \ \Upsilon \ &= \ \bot \end{aligned}$$

*Уплотнением* мульти-среды $\Upsilon$ называется среда $\overline{\Upsilon}$ такая, что

$$\forall v, e \in \Upsilon(v): \ \overline{\Upsilon}(v) \cong e.$$

Уплотнение соответствует “склеиванию” нескольких значений для каждой переменной в одно, в результате из мульти-среды получается обыкновенная среда.

### 2.2.2. Операция сопоставления с образцом

*Операция сопоставления* объекта $x$ с образцом $P$ в мульти-среде $\Upsilon$ обозначается $e\ \mathbf{match}_\Upsilon\ P$ и возвращает мульти-среду $\Upsilon'$ в случае успеха и $\perp$ в случае неудачи. Семантика этой операции представлена на Рис. 2.1.

$$\frac{\overline{\Upsilon}(v) = [e] \quad e \cong x}{x\ \mathbf{match}_\Upsilon\ <?v> = \Upsilon \uplus \{v \mapsto [x]\}}\ \text{match-var-e}$$

$$\frac{\overline{\Upsilon}(v) = <P>}{x\ \mathbf{match}_\Upsilon\ <?v> = (x\ \mathbf{match}_\Upsilon\ P) \Uparrow \{v \mapsto [x]\}}\ \text{match-var-P}$$

$$\frac{\mathbf{template}\,(T <p_1, \ldots, p_n> \{B\}) \quad \Upsilon' = (\biguplus_i \{p_i \mapsto <a_i>\}) \uplus \Upsilon}{x\ \mathbf{match}_\Upsilon\ <T\ a_1, \ldots, a_n> = x\ \mathbf{match}_{\Upsilon'}\ B}\ \text{match-app}$$

$$\frac{x = C@id\,\{f_i = v_i\}}{x\ \mathbf{match}_\Upsilon\ <? : C> = \Upsilon}\ \text{match-wc} \qquad \frac{x\ \mathbf{match}_\Upsilon\ <? : C> = \Upsilon}{x\ \mathbf{match}_\Upsilon\ <? : C^?> = \Upsilon}\ \text{match-wc-?}$$

$$\frac{}{NULL\ \mathbf{match}_\Upsilon\ <? : C^?> = \Upsilon}\ \text{match-null}$$

$$\frac{x = C@id\,\{f_i = v_i\} \quad v_i\ \mathbf{match}_\Upsilon\ P_i = \Upsilon_i}{x\ \mathbf{match}_\Upsilon\ \mathcal{TC}\,(C)\,@id'\,\{f_i = P_i\} = (\biguplus_i \Upsilon_i) \uplus \Upsilon}\ \text{match-ds}$$

$$\frac{x = [x_1, \ldots, x_n] \quad P = [P_1, \ldots, P_m]}{x\ \mathbf{match}_\Upsilon\ P = matchList_\Upsilon(x,\ P)}\ \text{match-list}$$

$$\frac{x = \{x_1, \ldots, x_n\} \quad P = \{P_1, \ldots, P_m\}}{x\ \mathbf{match}_\Upsilon\ P = matchSet_\Upsilon(x,\ P)}\ \text{match-set}$$

$$\frac{x \text{ — значение примитивного типа}}{x\ \mathbf{match}_\Upsilon\ x = \Upsilon}\ \text{match-prim}$$

Рис. 2.1: Семантика операции сопоставления с образцом

Согласно правилам *match-var-e*, *match-var-e* и *match-app* мульти-среда может хранить два вида значений: первоначально в нее помещается образец, соответствующий данной переменной (обозначается $<P>$), а когда этот шаблон применяется первый раз, значение заменяется соответствующим объектом (обозначается $[e]$). В дальнейшем мульти-среда пополняется объектами, структурно эквивалентными данному.

Задача сопоставления с образцом хорошо изучена. В частности

известно, что она NP-полна, если в образцах разрешается использовать переменные и ассоциативно-коммутативные операции (в нашем случае таким операциям эквивалентны неупорядоченные коллекции, то есть множества) [**?**]. Таким образом, алгоритм сопоставления имеет экспоненциальную трудоемкость, однако, как показала практика использования GRAMMATIC, на практике образцы, для которых сопоставление работает долго, не встречаются. Для полноты изложения мы приводим ниже краткое описание операции сопоставления для списков и множеств.

Основная вычислительная сложность операции сопоставления лежит в алгоритмах $matchList_{\Upsilon}(x,\, P)$ и $matchSet_{\Upsilon}(x,\, P)$. Функция $matchList_{\Upsilon}(x,\, P)$ использует перебор с возвратами для того, чтобы корректно обрабатывать подстановочные знаки кратности “+” и “*”. Обнаруживая такой подстановочный знак, функция $matchList(,\,)$ “пробует” сопоставить ему 0, 1, 2 и т. д. элементов списка, вызывая себя рекурсивно на оставшихся данных. Если одно из сопоставлений прошло успешно, перебор заканчивается.

Идея алгоритма достаточно проста, и чтобы не загромождать текст деталями, мы приводим только его схематичное описание для случая непустых списков. Наше описание не полностью учитывает обработку переменных: мы рассматриваем только случай, когда очередной элемент образца является подстановочным знаком, но он может являться ссылкой на переменную, которой сопоставлен подстановочный знак. В этом случае необходимо запомнить, что в случае удачного сопоставления в мульти-среду нужно записать соответствующее значение для данной переменной. Этот технический момент не меняет сути алгоритма, поэтому он не отражен в следующем псевдокоде, описывающим основные шаги алгоритма сопоставления списков:

$(x_h : x_t) \leftarrow x$ { $x_h$ — голова списка, $x_t$ — остальные элементы}
$(P_h : P_t) \leftarrow P$

**if** $P_h$ — подстановочный знак кратности $m \in \{+, *\}$ **then**
  $n_0 \leftarrow \begin{cases} 1, & \text{если } m = + \\ 0, & \text{если } m = * \end{cases}$
  **for all** $n \leftarrow n_0$ до $length(x)$ **do**
    $tail \leftarrow$ список $x$, кроме первых $n$ элементов
    $\Upsilon' \leftarrow matchList_{\Upsilon}(tail,\ P_t)$
    **if** $\Upsilon' \neq \perp$ **then**
      **return** $\Upsilon'$
  **return** $\perp$
**else**
  $\Upsilon' = x_h\ \mathbf{match}_{\Upsilon}\ P_h$
  **return** $matchList_{\Upsilon'}(x_t,\ P_t)$

Функция $matchSet_{\Upsilon}(x,\ P)$ несколько сложнее, поскольку порядок сопоставления не зафиксирован, а из-за наличия переменных его нельзя взять произвольными: от того, какая переменная будет сопоставлена первой, может зависеть, удастся ли найти значения для остальных. Задача сопоставления в этом случае сводится к перебору максимальных паросочетаний в двудольном графе [**?**]. Граф $G$ строится следующим образом: одну долю образуют элементы множества $x$, другую — $P$, элементы $x_i$ и $P_j$ соединены ребром, если $x_i\ \overline{\mathbf{match}}_{\Upsilon}\ P_j \neq \perp$. Операция $\overline{\mathbf{match}}$ игнорирует переменные, подставляя на их место соответствующие им образцы. Общая схема дальнейшей работы такова: если максимальное паросочетание связывает все элементы множества $x$, то есть шанс, что, перебирая такие паросочетания (их может быть несколько), и выполняя сопоставление согласно ребрам в них, мы найдем значения для всех переменных. Это отражено в следующем псевдокоде:

$M_0 \leftarrow$ максимальное паросочетание в $G$
**if** $|M_0| < |x|$ **then**
  **return** $\perp$

**for all** $M \leftarrow$ максимальное паросочетание в $G$ **do**
  {Попытка сопоставления согласно $M$ }
  $\Upsilon' \leftarrow \Upsilon$
  {Перебор ребер в паросочетании}
  **for all** $(x_i, P_j) \leftarrow M$ **do**
    $\Upsilon' \leftarrow x_i \ \mathbf{match}_{\Upsilon'} \ P_j$
  **if** $\Upsilon' \neq \perp$ **then**
    **return** $\Upsilon'$ {Сопоставление удачно}
**return** $\perp$

## 2.3. Семантика применения аспектов

**Определение 2.2.** *Аспектным правилом* (или *атомарным аспектом*) называется тройка $\mathcal{R} = \langle P, T, V \rangle$, где

- $P$ — образец (срез), связывающий переменные $v_1, \ldots, v_m$ и не имеющий свободных переменных;
- $T$ — наиболее конкретный тип элемента, который может быть успешно сопоставлен с $P$, то есть тип $P$ (см. Рис. 1.4);
- $V$ — набор *правил замены*, то есть кортежей $\langle v_i, t_i \rangle$, где
  - $v_i$ — переменная;
  - $t_i$ — шаблонное выражение, содержащее ссылки на переменные $v_1, \ldots, v_m$ как на шаблонные параметры.

*Аспект* представляет собой множество аспектных правил. Применение аспекта к грамматике сводится к последовательному применению составляющих его аспектных правил. Каждое правило применяется следующим образом: последовательно перебираются все объекты $e$ типа $T$, представленные в данной грамматике, на каждом из них происходит *применение* аспектного правила (обозначается $\mathcal{R}@e$), определенное ниже.

Для определения применения аспектного правила, нам потребуется операция *подстановки*, которая заменяет одни объекты внутри модели другими. Элементарная подстановка заменяет всего один объект и обозначается $x_1 \mapsto x_2$. Применение подстановки $\sigma$ к модели $m$ обозначается $(\sigma) \triangleright m$ и для элементарного случая $x_1 \mapsto x_2$ определяется следующим образом: все ссылки на объект $x_1$ заменяются ссылками на $x_2$. Неэлементарные подстановки строятся с помощью операции *композиции*: $(\sigma_1 \sqcup \sigma_2) \triangleright m$ соответствует последовательному применению двух подстановок $(\sigma_2) \triangleright ((\sigma_1) \triangleright m)$.

**Замечание 2.2.** *Композиция подстановок в общем случае не коммутативна, что приводит к проблеме, взаимодействия советов (advice interaction), присущей всем аспектно-ориентированным языкам (см., например, [29], где эта проблема рассмотрена очень подробно). Ниже мы введем в язык аспектов некоторые расширения, которые позволят в той или иной степени справиться с этой проблемой.*

Теперь определим операцию применения аспектного правила. Пусть $P$ **match** $e = \Upsilon \neq \bot$,
$V = \{\langle v_i, t_i \rangle \mid i = 1..m\}$, $\Upsilon(v_i) = [e^i_1, \ldots, e^i_{n_i}]$, тогда

$$\mathcal{R}@e = \left( \bigsqcup_{i=1}^{m} \bigsqcup_{j=1}^{n_i} e^i_j \mapsto \mathcal{I}_{\overline{\Upsilon}}[t_i] \right) \triangleright e$$

Фактически, результат сопоставления образца преобразуется в подстановку, заменяющую все значения, сопоставленные каждой переменной, результатами разворачивания соответствующих шаблонов, где средой является уплотнение результата сопоставления.

**Замечание 2.3.** *Определение аспектного правила, которое мы ввели, непосредственно соответствует советам с ключевым словом* `instead`*, поскольку происходит только замена. Ниже мы покажем, как реализовать другие два типа советов.*

## 2.4. Система типов

Применение подстановки может нарушить структуру модели, например, тип заменяющего объекта не удовлетворяет требованиям мета-модели для некоторой ссылки на заменяемый объект. Для того, чтобы исключить такие случаи, для аспектных правилах используется система типов, базирующаяся на системе типов для шаблонов, приведенной на Рис. 1.4. Подстановочные знаки, которые не учитываются правилами Рис. 1.4, типизируются следующим образом:

$$\frac{}{\vdash <? : \tau> : \tau}\ \text{wcard}$$

Теперь мы можем выписать правило, ограничивающее соотношение типов переменных в аспектном правиле и сопоставляемых им шаблонов. Для этого нам понадобится определить функцию $\Gamma(p)$, которая по образцу $p$ строит контекст $\Gamma$, соответствующий Рис. 1.4. Эта функция определяется так: $\Gamma(p)$ возвращает множество всех параметров шаблонов, использованных внутри $p$, вместе с их типами $(v : \tau)$. С использованием этой функции типизация аспектных правил выглядит так:

$$\frac{\mathcal{R} = \langle p, T, V\rangle \quad \Gamma(p) \vdash v : \tau \quad \Gamma(p) \vdash t : \sigma \quad \sigma \preceq \tau}{(\textbf{instead} <?v> : t) \in \mathit{Allowed}(\mathcal{R})}\ \text{aspect}$$

Здесь $\mathit{Allowed}(\mathcal{R})$ обозначает множество всех советов, которые разрешены к использованию в $V$. Данное правило требует, чтобы тип результата шаблонного выражения (в контексте $\Gamma(p)$) был подтипом типа соответствующей переменной. Поскольку переменную можно заменить любым объектом того же типа, не нарушив структурной корректности, все такие правила допустимы.

**Лемма 2.4.** *Если* $e$ **match** $p \neq \bot$, $\Psi, \mathfrak{M} \Vdash e : \tau$ *и* $\vdash p : \rho$, *то* $\tau \preceq \rho$.

*Доказательство.* Для доказательства необходимо рассмотреть случаи для возможных значений $\rho$ и показать, что сопоставление может

быть успешным только если $\tau$ является подтипом $\rho$. Здесь мы рассмотрим только случай $\rho = C$ в качестве примера.

Последним в дереве вывода $\vdash p : C$ может быть одно из следующих правил:

*ds-type(C)* — при сопоставлении применялось правило *match-ds*, следовательно $\Psi, \mathfrak{M} \Vdash e : C$, а $C \preceq C$;

*wc* — применялось правило *match-wc*, рассуждения аналогичны предыдущему случаю;

*subtype* — задача сводится к аналогичной при $\sigma \preceq C$, по транзитивности $\preceq$ утверждение леммы верно;

*appl* — этому правилу предшествует правило *abstr*, требующее, чтобы тело шаблона $b$ имело тип $\sigma$, то есть задача сводится к аналогичному доказательству утверждения для выражения $b$;

*var* — если при сопоставлении применялось правило *match-var-P*, то тип $P$ может быть только подтипом $C$, а если *match-var-e*, то $e \cong e'$, где для сопоставления $e'$ применялось правило *match-var-P*.

В остальных случаях доказательство аналогично. □

**Лемма 2.5.** *Если* $e$ **match** $p = \Upsilon \neq \bot$, $\Gamma(e) \vdash v : \tau$ *и* $x \in \Upsilon(v)$, *то* $\Psi, \mathfrak{M} \Vdash x : \tau$.

*Доказательство.* Значения переменных помещаются в $\Upsilon$ в правилах *match-var-P* и *match-var-e*. В первом случае корректность типа значения следует из предыдущей леммы, так как по правилу *appl* $P$ будет корректно типизрован. Во втором случае результат $x \cong e$, где для $e$ применялось правило *match-var-P*. □

**Лемма 2.6.** *Если* $e$ **match** $p = \Upsilon \neq \bot$ *и* $\Gamma(p) \vdash t : \tau$, *то*

$\Psi, \mathfrak{M} \Vdash \mathcal{I}_{\overline{\Upsilon}}[t] : \tau$.

*Доказательство.* Из предыдущей леммы следует, что среда $\overline{\Upsilon}$ согласована с контекстом $\Gamma(p)$; отсюда по теореме 2.2 о сохранении типов, теореме 2.3 о нормализации и лемме 2.1 о нормальных формах следует утверждение данной леммы. □

**Определение 2.3.** Элементарная подстановка $x \mapsto y$ называется *безопасной* в мета-модели $\mathfrak{M}$, если в данной мета-модели любая ссылка, которая может указывать на $x$, может указывать также и на $y$.

Неэлементарная подстановка называется безопасной, если она получена композицией безопасных подстановок.

**Теорема 2.7.** *Если $\mathcal{R}$ таково, что $V \subseteq Allowed(\mathcal{R})$, то в результате применения $\mathcal{R}@e$ не нарушаются структурные ограничения, накладываемые мета-моделью.*

*Доказательство.* Согласно определению операции применения, данное утверждение можно переформулировать так: подстановка

$$\bigsqcup_{i=1}^{m} \bigsqcup_{j=1}^{n_i} e_j^i \mapsto \mathcal{I}_{\overline{\Upsilon}}[t_i]$$

является безопасной. Данное утверждение следует из трех предыдущих лемм. □

### 2.4.1. Контроль над недетерминированным поведением

В общем случае результат применения аспекта может сильно зависеть от порядка применения аспектных правил, что затрудняет написание сложных аспектов. Как было отмечено выше, эта проблема полностью не решена на только для случая произвольного языка, но и для конкретных аспектно-ориентированных языков программирования. В настоящем разделе мы рассмотрим условия, при которых порядок подстановки не важен, а также опишем простые средства, с помощью которых применение аспектов можно сделать более предсказуемым.

**Определение 2.4.** Класс $C$ в мета-модели $\mathfrak{M}$ называется *допускающим локальную замену*, если в данной мета-модели все ссылки, имеющие типом этот класс и все его подклассы и суперклассы, являются агрегирующими.

Например, в целевой мета-модели GRAMMATIC класс `Expression` и все его подклассы допускают локальную замену. По сути, это свойство означает, что для замены объекта класса $C$ при подстановке достаточно изменить всего одну ссылку в модели, поскольку всякий объект может одновременно указывать не более одной агрегирующей ссылки.

**Определение 2.5.** Подстановки $x \mapsto y$ и $z \mapsto w$ называются *совместимыми*, если $y$ и $z$ — разные объекты и $x$ и $w$ — разные объекты.

Композиция совместимых подстановок называется *правильной*.

**Лемма 2.8.** *Пусть класс $C$ допускает локальную замену, тогда для любого объекта $x$ этого класса, безопасной подстановки $x \mapsto y$ и совместимой с ней безопасной подстановкой $\sigma$, операция композиции допускает перемену мест аргументов:*

$$x \mapsto y \sqcup \sigma \equiv \sigma \sqcup x \mapsto y$$

*Доказательство.* Введем обозначения:

$$\alpha := x \mapsto y \sqcup \sigma$$

$$\beta := \sigma \sqcup x \mapsto y$$

Две подстановки эквивалентны, если их применение к одной и той же модели всегда дает один и тот же результат. Пусть модель $m$ не содержит $x$, тогда

$$(\alpha) \triangleright m = (\beta) \triangleright m = (\sigma) \triangleright m,$$

поскольку из-за требования совместимости $\sigma$ не может добавить $x$ в модель. Если $m$ содержит $x$, то на него есть не более одной ссылки,

то есть объекты, которые появляются после подстановок, не имеют ссылок на $x$. Следовательно, результат обеих подстановок будет одинаков. □

**Лемма 2.9.** *Все элементарные подстановки, составляющие*

$$\bigsqcup_{i=1}^{m} \bigsqcup_{j=1}^{n_i} e_j^i \mapsto \mathcal{I}_{\overline{\Upsilon}}[t_i]$$

*попарно совместимы.*

*Доказательство.* Результат применения шаблона всегда является новым объектом и, следовательно, не может совпадать ни с одним из объектов в левой части подстановок. □

**Теорема 2.10.** *Результат применения аспектного правила, оперирующего только объектами классов допускающих локальную замену, не зависит от порядка объединения элементарных подстановок.*

*Доказательство.* Утверждение теоремы следует из двух предыдущих лемм. □

Если условия теоремы не выполняются для данного языка, контроль за разрешением неоднозначностей предоставляется программисту. Так порядок применения аспектных правил соответствует порядку их следования в описании аспекта, а порядок применения советов — порядку их следования в аспектном правиле.

Кроме того, во время применения аспектов осуществляются проверки, и в случае обнаружения неоднозначности генерируются предупреждения. Дополнительно можно использовать механизм контроля кратности сопоставления: указывать минимальное и максимальное число объектов, которые могут быть сопоставлены данному срезу или переменной. Если ограничения на кратность нарушаются, также генерируется предупреждение.

## § 3. Пример: расширение простого языка

В качестве примера применения описанного выше метода в данном разделе описано расширение простого предметно-ориентированного языка TOY. Данный язык позволяет описывать интерфейсы классов, например:

```
class Iterator {
    fun hasNext() : Boolean;
    fun next() : Object;
}
class Collection {
    fun add(obj : Object) : Boolean;
    fun iterator() : Iterator;
}
```

Как видно из примера, классы имеют имена, а в теле содержат объявления функций. Грамматика данного языка, приведенная в Лист. 3.1 позволяет также указывать суперкласс после ключевого слова `extends`.

```
specification : class*;
class
    : 'class' name ('extends' name)? '{'
        function*
      '}'
    ;
function
    : 'fun' name '(' <List parameter ','> ')' ':' type;
parameter
    : name ':' type;
type
    : name ('*' | '+' | '?')?;
name : NAME;
```

Листинг 3.1: Грамматика языка TOY

Целевая мета-модель данного языка приведена в Рис. 3.1 (слева) в текстовой нотации. Из названий классов видно, как они сопоставлены символам грамматики. На том же рисунке справа показан результат применения преобразования $\mathcal{TC}(\bullet)$ к классам целевой метамодели. Согласно описанной выше процедуре, грамматика расширена, и к каждому правилу добавлена продукция "`: term`". Соответствующее применение шаблона `Templates` (Лист. 1.1) выглядит следующим образом:

```
class Specification {
    val classes : Class*;
}
class Class {
    val name : Name;
    ref superclass : Class?;
    val functions : Function*;
}
class Function {
    val name : Name;
    val params : Parameter*;
    ref returnType : Class;
}
class Parameter {
    val name : Name;
    ref type : Class;
}
class Name {
    attr name : String;
}
```

```
class SpecificationT extends Term {
    val classes : Term*;
}
class ClassT extends Term {
    val name : Term;
    ref superclass : Term?;
    val functions : Term*;
}
class FunctionT extends Term {
    val name : Term;
    val params : Term*;
    ref returnType : Term;
}
class ParameterT extends Term {
    val name : Term;
    ref type : Term;
}
class NameT extends Term {
    attr name : Term;
}
```

Рис. 3.1: Классы целевой мета-модели исходного языка и языка шаблонов

```
<Templates
    ('Specification' | 'Class' | 'Function' | 'Parameter' | 'Name'),
    (specification, class, function, parameter, name)>
```

Полученный язык позволяет записывать следующие шаблонные выражения, достаточно близкие к шаблонным классам C++:

```
template Iterator<N : String, T : Class> : Class {
    class <?N> {
        fun hasNext() : Boolean
        fun next() : <?T>
    }
}
template Collection<N : String, IN : String, T : Class> : Class {
    class <?N> {
        fun contains(object : <?T>) : Boolean
        fun add(object : <?T>) : Boolean
        fun iterator() : <Iterator <?IN> <?T>>
    }
}
```

Важным отличием от шаблонов C++ является необходимость указывать имена как параметры шаблонов. Эту проблему можно устранить, если добавить к процедуре разворачивания шаблонов стадию переименования, запрограммированную вручную. Добавление подобных функций до и после семантических операций, описанных выше, не пред-

ставляет затруднений и гораздо проще, чем реализация всего механизма шаблонов полностью вручную.

Полученный при расширении языка TOY механизм аспектов позволяет реализовать внешние декларации функций (аналогично Inter-Type Declarations в ASPECTJ) следующим образом:

```
on class <? : String> { ?funs=<? : Function*> } perform
    instead funs : <?funs>; fun toString() : String;
```

Данное аспектное правило добавляет в тело каждого класса функцию `toString()`, что демонстрирует возможности квантификации (все классы обрабатываются одним правилом) и незнания (классы не готовятся для расширения специальным образом и имеют смысле даже если аспект не применялся).

## § 4. Сравнение с существующими подходами

Результаты сравнения предложенного в данной главе подхода с описанными в литературе приведены таблицах 4.1 и 4.2. Шаблоны и аспекты рассматривались отдельно, поскольку, кроме проекта REUSEWARE, ни один подход не обеспечивает расширение языков обоими механизмами композиции. Детальное сравнение предложенного в данной главе подхода с подходом проекта REUSEWARE приведено в конце данного раздела.

Для сравнения возможностей по поддержке шаблонов (Таб. 4.1) были выбраны семь проектов. Сравнение проводилось по следующим критериям:

(а) наличие поддержки внешних ПОЯ; три из семи проектов базируются на конкретных языках программирования и пригодны, таким образом, только для разработки исполняемых внутренних предметно-ориентированных языков;

(б) поддержка текстовых языков (рассматривалось только для внешних ПОЯ);

| | Lisp | MacroML | Haskel TMP | CPP | Velocity и др. | MPS | Reuseware | *Данная работа* |
|---|---|---|---|---|---|---|---|---|
| *Ссылка* | [75] | [24] | [71] | [44] | [26] | [39] | [60] | |
| Поддержка внешних ПОЯ | - | - | - | + | + | + | + | + |
| Применимо для текстовых языков | n/a | n/a | n/a | + | + | - | + | + |
| Замкнутость | + | + | + | + | + | n/a | - | + |
| Контроль корректности | n/a | + | + | - | - | + | + | + |
| Логика в теле шаблона | + | + | + | +/- | + | + | - | - |
| Используемый термин | M | M | T | M | T | T | CF | T |

Таблица 4.1: Поддержка шаблонов

(в) *замкнутость* расширенного языка — синтаксическая интеграция механизма шаблонов с другими конструкциями языка;

(г) контроль корректности использования шаблонов, выполняемый *до* разворачивания — эта возможность очень важна, поскольку позволяет рано обнаруживать и быстро исправлять ошибки;

(д) поддержка в теле шаблона сложной логики (условий, циклов и т. д.).

Отдельной строкой в таблице указано, какой термин использует данный проект: “шаблон” (“T” от “template”) или макроопределение (“M” от “macro”); видно, что частота употребления терминов примерно совпадает[1].

Из таблицы видно, что только предложенный в данной работе подход позволяет расширять текстовые внешние ПОЯ шаблонами в замкнутой форме, при этом обеспечивая контроль корректности. Подходы основанные на конкретных языках программирования не позволяют создавать внешние ПОЯ. Препроцессор CPP и генераторы текста

[1]Проект Reuseware использует термин “component fragment” для обозначения того же понятия, в таблице это отражено аббревиатурой “CF”.

(VELOCITY и др.) не осуществляют контроль корректности до разворачивания шаблона. MPS не поддерживает текстовые языки, что создает необходимость в поддержке сложными инструментами и трудности в интеграции с другими средствами разработки, а REUSEWARE создает незамкнутый механизм шаблонов.

Результаты аналогичного сравнения для аспектов приведены в таблице Таб. 4.2. Сравнение проводилось последующим критериям:

| | JAMI | ASPECTJ | XCPL | REUSEWARE | ASPECT.NET | POPART | XASPECTS | VAN WYK'03 | STRATEGO/XT | *Данная работа* |
|---|---|---|---|---|---|---|---|---|---|---|
| *Ссылка* | [29] | [62] | [17] | [60] | [69] | [20] | [72] | [78] | [10] | |
| Поддержка многих языков | + | - | + | + | + | + | + | + | + | + |
| Независимость от платф. | - | - | + | + | - | + | - | + | + | + |
| Неисполняемые языки | - | - | + | + | - | - | - | + | + | + |
| Автоматизация | n/a | - | - | + | + | - | + | + | - | + |
| Незнание | + | + | + | - | + | + | + | + | + | + |
| Спец. язык срезов | - | + | + | - | + | + | + | - | + | + |
| Семантика встраивания | + | + | - | + | + | + | + | + | + | + |
| Сумма | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 7 |

Таблица 4.2: Поддержка аспектов

(a) поддержка рассматриваемой технологией многих языков как общего назначения, так и предметно-ориентированных; отдельно рассматривались независимость от платформы (здесь под платформой понимается аппаратно-программная среда, то есть виртуальные машины (JAVA, .NET) тоже считаются платформами, поддержка неисполняемых языков — ярким примером такого языка GRAMMATIC, поддержка аспектов в котором весьма полезна, автоматизация разработки — ряд работ предлагает методики, основанные на тех или иных парадигмах программирования, не

автоматизирующих разработку, а предлагающих более удобный способ ручной разработки;

(б) поддержка аспектами незнания;

(в) наличие специализированного языка срезов: в работе [78] срезы предлагается реализовывать как функции на языке HASKELL, что затрудняет их написание и, давая бóльшие возможности, увеличивает вероятность ошибки;

(г) описывается ли в рамках подхода семантика встраивания советов.

Из таблицы видно, что только предложенный в данной работе подход поддерживает все указанные возможности.

Как было сказано выше, подход проекта REUSEWARE [60] наиболее близок к предложенному в данной главе, поскольку он также обладает следующими важными преимуществами: (а) обеспечивает структурную корректность результатов, (б) используем единый формализм для описания шаблонов и аспектов. Также как и предложенный подход, REUSEWARE не поддерживает условных операторов и циклов в теле шаблона. Недостатками REUSEWARE по сравнению с нашим подходом являются

- незамкнутость шаблонов — базовый язык шаблонов не интегрируется с расширяемым языком и требует отдельной инструментальной поддержки,
- отсутствие незнания и языка срезов в аспектах — точки встраивания должны быть специальным образом подготовлены для расширения и явно отмечены.

## § 5. Выводы

1. Разработана процедура расширения предметно-ориентированных языков механизмом шаблонов.

2. Для данного механизма формализована семантика и разработана система типов, гарантирующая структурную корректность результатов разворачивания шаблонов.

3. С использованием тех же базовых понятий разработана процедура расширения ПОЯ механизмом аспектов.

4. Описаны семантика сопоставления с образцом и применения аспектных правил. Показано что предложенная система типов гарантирует безопасность подстановок.

# Список литературы


[1] *Agesen, O.* Adding type parameterization to the java language / O. Agesen, S. N. Freund, J. C. Mitchell // *SIGPLAN Not.* — 1997. — Vol. 32, no. 10. — Pp. 49–65.

[2] *Aho, A. V.* Compilers: principles, techniques, and tools / A. V. Aho, R. Sethi, J. D. Ullman. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] *Aksit, M.* Compiler generation based on grammar inheritance. — 1990. http://doc.utwente.nl/19862/.

[4] Apache Derby. — http://db.apache.org/derby/.

[5] The asf+sdf meta-environment: A component-based language development environment / M. G. J. van den Brand, A. v. Deursen, J. Heering et al. // CC '01: Proceedings of the 10th International Conference on Compiler Construction. — London, UK: Springer-Verlag, 2001. — Pp. 365–370.

[6] Aspect-oriented programming / G. Kiczales, J. Lamping, A. Mendhekar et al. // In ECOOP. — SpringerVerlag, 1997.

[7] *A■mann, U.* Invasive Software Composition / U. A■mann. — Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[8] *Atkinson, C.* Model-driven development: A metamodeling foundation / C. Atkinson, T. Kühne // *IEEE Softw.* — 2003. — Vol. 20, no. 5. — Pp. 36–41.

[9] *Baader, F.* Term rewriting and all that / F. Baader, T. Nipkow. — New York, NY, USA: Cambridge University Press, 1998.

[10] *Bagge, A.* DSAL = library+notation: Program transformation for domain-specific aspect languages / A. Bagge, K. K. T. // First Domain Specific Aspect Language Workshop (DSAL'06). — 2006.

[11] *Berners-Lee, T.* Uniform resource identifiers (uri) generic syntax: Tech. Rep. RFC 2396 / T. Berners-Lee, R. Fielding, L. Masinter: 1998. — http://www.ietf.org/rfc/rfc2396.txt.

[12] *Booch, G.* Object-Oriented Analysis and Design with Applications (3rd Edition) / G. Booch. — Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[13] *Booch, G.* Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series) / G. Booch, J. Rumbaugh, I. Jacobson. — Addison-Wesley Professional, 2005.

[14] *Budinsky, F.* Eclipse Modeling Framework / F. Budinsky, S. A. Brodsky, E. Merks. — Pearson Education, 2003.

[15] *Ciric, M. D.* Parsing in different languages / M. D. Ciric, S. R. Rancic // *FACTA UNIVERSITATIS*. — 2005. — Vol. 18, no. 2. — Pp. 299–307.

[16] The compost, compass, inject/j and recoder tool suite for invasive software composition: Invasive composition with compass aspect-oriented connectors. / D. Heuzeroth, U. A■mann, M. Trifu, V. Kuttruff // GTTSE / Ed. by R. Lämmel, J. Saraiva, J. Visser. — Vol. 4143 of *Lecture Notes in Computer Science*. — Springer, 2006. — Pp. 357–377.

[17] *Dekel, U.* Towards a standard family of languages for matching patterns in source code / U. Dekel, T. Cohen, S. Porat // SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering. — Washington, DC, USA: IEEE Computer Society, 2003. — P. 10.

[18] *Denny, J. E.* Ielr(1): practical lr(1) parser tables for non-lr(1) grammars with conflict resolution / J. E. Denny, B. A. Malloy // SAC '08: Proceedings of the 2008 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2008. — Pp. 240–245.

[19] Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. — Addison-Wesley, 1995.

[20] *Dinkelaker, T.* Untangling crosscutting concerns in domain-specific languages with domain-specific join points / T. Dinkelaker, M. Monperrus, M. Mezini // DSAL '09: Proceedings of the 4th workshop on Domain-specific aspect languages. — New York, NY, USA: ACM, 2009. — Pp. 1–6.

[21] Eli: a complete, flexible compiler construction system / R. W. Gray, S. P. Levi, V. P. Heuring et al. // *Communications of the ACM*. — 1992. — Vol. 35, no. 2. — Pp. 121–130.

[22] *Fowler, M.* Domain specific languages. — http://martinfowler.com/dslwip/. — 2010.

[23] *Gagnon, E. M.* SableCC, an object-oriented compiler framework / E. M. Gagnon, L. J. Hendren // TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems. — Washington, DC, USA: IEEE Computer Society, 1998. — Pp. 140–154.

[24] *Ganz, S. E.* Macros as multi-stage computations: type-safe, generative, binding macros in macroml / S. E. Ganz, A. Sabry, W. Taha // ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming. — New York, NY, USA: ACM, 2001. — Pp. 74–85.

[25] *Gärtner, F.* The PretzelBook. — http://www.informatik.tu-darmstadt.de/BS/Gaertner/pretzel/dist 1998.

[26] *Gradecki, J. D.* Mastering Apache Velocity / J. D. Gradecki, J. Cole. — Wiley, 2003.

[27] *Grimm, R.* Better extensibility through modular syntax / R. Grimm // *SIGPLAN Not.* — 2006. — Vol. 41, no. 6. — Pp. 38–51.

[28] Haskell 98 Language and Libraries: The Revised Report / Ed. by S. P. Jones. — http://haskell.org/, 2002. — September. — P. 277. http://haskell.org/definition/haskell98-report.pdf.

[29] *Havinga, W.* Prototyping and composing aspect languages / W. Havinga, L. Bergmans, M. Aksit // ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming. — Berlin, Heidelberg: Springer-Verlag, 2008. — Pp. 180–206.

[30] *Hedin, G.* JastAdd: an aspect-oriented compiler construction system / G. Hedin, E. Magnusson // *Science of Computer Programming*. — 2003. — Vol. 47, no. 1. — Pp. 37–58.

[31] *Henry, K.* A crash overview of groovy / K. Henry // *Crossroads*. — 2006. — Vol. 12, no. 3. — Pp. 5–5.

[32] Human-usable textual notation. — http://www.omg.org/spec/hutn/.

[33] Implementation of multiple attribute grammar inheritance in the tool lisa / M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević // *SIGPLAN Not.* — 1999. — Vol. 34, no. 6. — Pp. 68–75.

[34] *Ingalls, D. H. H.* The smalltalk-76 programming system design and implementation / D. H. H. Ingalls // POPL '78: Proceedings of the 5th

ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — New York, NY, USA: ACM, 1978. — Pp. 9–16.

[35] Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. — 2nd edition. — The MIT Press, 2001.

[36] *ISO*. ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF / ISO. — 1996. — P. 12. http://www.iso.ch/cate/d26153.html.

[37] ISO/IEC 9075:1992, Database Language SQL. — http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. — 1992.

[38] The Java Language Specification, Third Edition / J. Gosling, B. Joy, G. Steele, G. Bracha. — 3 edition. — Amsterdam: Addison-Wesley Longman, 2005. — June. — P. 688.

[39] *JetBrains*. Meta Programming System (MPS). — http://www.jetbrains.com/mps. — 2009.

[40] *Johnson, S. C.* Yacc: Yet another compiler-compiler: Tech. rep. / S. C. Johnson: Bell Laboratories, 1979.

[41] *Jouault, F.* Km3: A dsl for metamodel specification / F. Jouault, J. Bézivin // FMOODS / Ed. by R. Gorrieri, H. Wehrheim. — Vol. 4037 of *Lecture Notes in Computer Science*. — Springer, 2006. — Pp. 171–185.

[42] *Jouault, F.* Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering / F. Jouault, J. Bézivin, I. Kurtev // GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering. — New York, NY, USA: ACM, 2006. — Pp. 249–254.

[43] *Kastens, U.* Ordered attributed grammars / U. Kastens // *Acta Informatica*. — 1980. — Vol. 13, no. 3. — Pp. 229–256.

[44] *Kernighan, B. W.* The C programming language / B. W. Kernighan, D. M. Ritchie. — Prentice-Hall, Englewood Cliffs, N.J. :, 1978. — Pp. x, 228 p. ;.

[45] *Khedker, U.* Data Flow Analysis: Theory and Practice / U. Khedker, A. Sanyal, B. Karkare. — Boca Raton, FL, USA: CRC Press, Inc., 2009.

[46] *Klint, P.* Toward an engineering discipline for grammarware / P. Klint, R. Lämmel, C. Verhoef // *ACM Trans. Softw. Eng. Methodol.* — 2005. — Vol. 14, no. 3. — Pp. 331–380.

[47] *Klint, P.* Term rewriting meets aspect-oriented programming. / P. Klint, T. van der Storm, J. J. Vinju // Processes, Terms and Cycles / Ed. by A. Middeldorp, V. van Oostrom, F. van Raamsdonk, R. C. de Vrijer. — Vol. 3838 of *Lecture Notes in Computer Science*. — Springer, 2005. — Pp. 88–105.

[48] *Knuth, D. E.* Semantics of context-free languages / D. E. Knuth // *Theory of Computing Systems*. — 1968. — June. — Vol. 2, no. 2. — Pp. 127–145.

[49] *Kodaganallur, V.* Incorporating language processing into java applications: A JavaCC tutorial / V. Kodaganallur // *IEEE Software*. — 2004. — Vol. 21, no. 4. — Pp. 70–77.

[50] *Lämmel, R.* Recovering Grammar Relationships for the Java Language Specification / R. Lämmel, V. Zaytsev // Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. — IEEE, 2009. — Сентябрь. — Pp. 178–186. — Full version submitted for journal publication.

[51] *Latendresse, M.* Regreg:a lightweight generator of robust parsers for irregular languages / M. Latendresse // *Reverse Engineering, Working Conference on.* — 2003. — Vol. 0. — P. 206.

[52] *Levine, J.* Flex & Bison / J. Levine. — O'Reilly, 2009.

[53] *McPeak, S. G.* Elkhound: A fast, practical glr parser generator: Tech. rep. / S. G. McPeak. — Berkeley, CA, USA: 2003.

[54] Meta-object facility. — http://www.omg.org/mof/.

[55] Model driven architecture. — http://www.omg.org/mda/.

[56] *Moon, D. A.* Programming language for old timers. — http://users.rcn.com/david-moon/PLOT.

[57] *Mossenbock, H.* Coco/r - a generator for fast compiler front ends: Tech. rep. / H. Mossenbock: 1990.

[58] *Nickel, U.* The fujaba environment / U. Nickel, J. Niere, A. Zündorf // ICSE '00: Proceedings of the 22nd international conference on Software engineering. — New York, NY, USA: ACM, 2000. — Pp. 742–745.

[59] *Odersky, M.* Programming in Scala: [a comprehensive step-by-step guide] / M. Odersky, L. Spoon, B. Venners. — artima, 2008. — Vol. 1. ed., version 5. — P. 736.

[60] On language-independent model modularisation / F. Heidenreich, J. Henriksson, J. Johannes, S. Zschaler. — 2009. — Pp. 39–82.

[61] Open fortran parser. — http://fortran-parser.sourceforge.net.

[62] An overview of AspectJ / G. Kiczales, E. Hilsdale, J. Hugunin et al. // ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming. — London, UK: Springer-Verlag, 2001. — Pp. 327–353.

[63] *Parr, T.* The Definitive ANTLR Reference: Building Domain-Specific Languages / T. Parr. Pragmatic Programmers. — First edition. — Pragmatic Bookshelf, 2007. — Май.

[64] *Parr, T. J.* Antlr: A predicated- :::: Ll(k) :::: Parser generator / T. J. Parr, R. W. Quong // *SPE*. — 1995. — Vol. 25, no. 7. — Pp. 789–810.

[65] *Pierce, B. C.* Types and programming languages / B. C. Pierce. — Cambridge, MA, USA: MIT Press, 2002.

[66] PostgreSQL. — http://www.postgresql.org/.

[67] *Rebernak, D.* A tool for compiler construction based on aspect-oriented specifications / D. Rebernak, M. Mernik // COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 11–16.

[68] Revised report on the algorithmic language algol 68 / A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck et al. // *Acta Inf.* — 1975. — Vol. 5. — Pp. 1–236.

[69] *Safonov, V. O.* Using Aspect-Oriented Programming for Trustworthy Software Development / V. O. Safonov. — New York, NY, USA: Wiley-Interscience, 2008.

[70] Separation of concerns in compiler development using aspect-orientation / X. Wu, B. R. Bryant, J. Gray et al. // SAC '06: Proceedings of the 2006 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2006. — Pp. 1585–1590.

[71] *Sheard, T.* Template meta-programming for haskell / T. Sheard, S. P. Jones // Haskell '02: Proceedings of the 2002 ACM SIGPLAN

workshop on Haskell. — New York, NY, USA: ACM, 2002. — Pp. 1–16.

[72] *Shonle, M.* Xaspects: an extensible system for domain-specific aspect languages / M. Shonle, K. Lieberherr, A. Shah // OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. — New York, NY, USA: ACM, 2003. — Pp. 28–37.

[73] Silver: an extensible attribute grammar system / E. Van Wyk, D. Bodin, J. Gao, L. Krishnan // *ENTCS*. — 2008. — Vol. 203, no. 2. — Pp. 103–116.

[74] *Skalski, K.* Meta-programming in Nemerle. — http://nemerle.org/metaprogramming.pdf. — 2004.

[75] *Steele Jr., G. L.* Common LISP: the language / G. L. Steele, Jr. — Newton, MA, USA: Digital Press, 1984.

[76] Stratego/xt 0.17. a language and toolset for program transformation / M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser // *Sci. Comput. Program.* — 2008. — Vol. 72, no. 1-2. — Pp. 52–70.

[77] *Stroustrup, B.* The C++ Programming Language / B. Stroustrup. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[78] *Van Wyk, E.* Implementing aspect-oriented programming constructs as modular language extensions / E. Van Wyk // *Sci. Comput. Program.* — 2007. — Vol. 68, no. 1. — Pp. 38–61.

[79] *Varró, D.* Metamodeling mathematics: A precise and visual framework for describing semantics domains of uml models / D. Varró, A. Pataricza // UML '02: Proceedings of the 5th

International Conference on The Unified Modeling Language. — London, UK: Springer-Verlag, 2002. — Pp. 18–33.

[80] Weaving a debugging aspect into domain-specific language grammars / H. Wu, J. Gray, S. Roychoudhury, M. Mernik // SAC '05: Proceedings of the 2005 ACM symposium on Applied computing. — New York, NY, USA: ACM, 2005. — Pp. 1370–1374.

[81] *Wö■, A.* LL(1) conflict resolution in a recursive descent compiler generator / A. Wö■, M. Löberbauer, H. Mössenböck // JMLC / Ed. by L. Böszörményi, P. Schojer. — Vol. 2789 of *Lecture Notes in Computer Science*. — Springer, 2003. — Pp. 192–201.

[82] Xml metadata interchange. — http://www.omg.org/xmi/.

[83] *Behrens, H.* Xtext User Guide. — http://www.eclipse.org/Xtext/documentation/0_7_2/xtext.pdf. — 2009.

[84] *Zdun, U.* A dsl toolkit for deferring architectural decisions in dsl-based software design / U. Zdun // *Information and Software Technology*. — 2010. — Vol. 52. — Pp. 733–748.