Курсовая работа
Руководитель: Вадим Гуров (кандидат наук, программист ООО "ИнтеллиДжей Лабс")
Интеграция языка для работы с реляционными базами данных в базовый язык JetBrains MPS

Выполнил Никитин Павел Антонович
Кафедра системного программирования СПбГУ

9 мая 2009

1 Введение

Для разработчиков программного обеспечения из разных доменов могут быть полезными всевозможные доменно-специфичные расширения языков программирования общего назначения. Например, разработчики приложений для банковских нужд оценят встроенную в язык поддержку работы с денежными единицами. К сожалению, традиционные текстовые языки обязаны обладать однозначной грамматикой, что делает их трудно расширяемыми. Целью данной работы было именно расширение языка общего назначения (Java) и добавление в него конструкций для работы с реляционными базами данных. Для решения проблемы возможной неоднозначности грамматики полученного языка был выбран языковой инструментарий JetBrains MPS (Meta Programming System) – средство для создания языков и интегрированная среда для разработки на них. MPS решает эту проблему, работая непосредственно с

абстрактным синтаксическим деревом программы, для редактирования которого используется текстово-подобный проекционный редактор. Однако, языково-ориентированное программирование в таком виде, как оно существует сегодня – достаточно молодая парадигма. Например, как другой заметный его представитель – Intentional Software, так и MPS выходят из beta только в этом году.

2 О работе с реляционными БД

Тем не менее, потребность удобной работы с базами данных возникла давно. Поэтому существует масса ad-hoc решений для различных языков программирования. Самый известный из них – так называемый Embedded SQL. При таком подходе программа должна быть обработана специальным препроцессором, прежде чем она будет откомпилирована компилятором базового языка программирования. Препроцессор распознаёт вызовы запросов внутри языковых предложений и преобразует их в библиотечные вызовы, также распознаются специальным образом оформленные ссылки на переменные базового языка внутри SQL-предложений и некоторые другие менее фундаментальные конструкции – т. е. происходит трансляция. Данный подход доступен для языков C, Ada, Cobol, Fortran.

Похожий подход предлагает и стандарт SQLJ для языка Java. В этом стандарте в качестве конечной библиотеки так или иначе используется JDBC, входящая в стандартную поставку от Sun начиная с JDK версии 1.1, но чаще всего она обёрнута в пропиетарный код, как, например, в реализации SQLJ от Oracle. Пример кода на SQLJ:

```
int i, j;
i = 1;
#sql {  SELECT field INTO :OUT j
          FROM table
          WHERE id = :IN i };
System.out.println(j);
```

О JDBC мы поговорим подробнее, так как именно она является основой для многих решений, работающих с базами, например Hibernate. В

принципе, для несложных приложений вполне применимо использование JDBC напрямую, без каких-либо обёрток, поэтому наряду с Embedded SQL будем рассматривать и решения, не использующие препроцессор.

Все эти решения объединяет одно – слабо выраженная структура. Тем не менее, они реально используются для разработки программного обеспечения, и в процессе общения с ними разработчик может столкнуться с вытекающими отсюда трудностями. Во-первых, это нарушение статического синтаксиса запроса, например – опечатка в ключевом слове SELECT. Класс решений с препроцессором позволет найти такую ошибку на стадии трансляции, что в принципе, приемлимо. Однако строки, явно попадающие в параметры функций JDBC не обрабатываются препроцессором и, следовательно, ошибка будет замечена только во время выполнения и приведёт к исключению SQLException, что неприемлимо даже при наличии покрытия тестами части кода, содержащей ошибку, так как отвлекает разработчика от решения актуальных задач. Во-вторых, это проблема вложенности запросов. В JDBC конструкция с вложением языков вида $Java_1[SQL_1[Java_2[SQL_2]]]$ может привести к трудно воспроизводимому нарушению синтаксиса формируемого запроса $SQL_1 \cup SQL_2$, если на уровне вложенности $Java_2$ используется нетривиальное ветвление. SQLJ же вовсе исключает возможность такой вложенности.

Ещё один интересный подход - создание, в отличие от SQLJ, не внешнего доменно-специфичного языка (со своим расширенным синтаксисом и другой грамматикой), а внутреннего, то есть использование средств базового языка (Java) для придания коду на нём подобия специального языка (для работы с реляционными БД в нашем случае). Данный подход реализуют такие библиотеки, как, например, jequel, squill, quaere. Пример кода на jequel:

```
final ARTICLE ARTICLE2 = ARTICLE.as("article2");
final Field OID = ARTICLE_C.OID.as("article_c_oid");

SqlString sql = select(ARTICLE2.OID, ARTICLE_C_OID)
                    .from(ARTICLE2, ARTICLE_C)
                    .where(ARTICLE2.OID.eq(OID));
```

Особняком стоит недоступное для Java-мира расширение языка C#

LINQ to SQL, но его использование предполагает знание своего SQL-подобного синтаксиса помимо знания LINQ и SQL.

3 Значимость IDE

Конечно, наличия ошибок в коде можно избежать при достаточной дисциплине со стороны разработчика. Но для существенного повышения производительности используются интегрированные среды разработки с структурированной подсказкой вводимого кода, подсветкой ошибок, анализом потока управления, рефакторингом и прочими полезными функциями. Однако популярные среды разработки на Java, такие, как Eclipse и IntelliJ IDEA не поддерживают Embedded SQL ни самостоятельно, ни в качестве плагинов. Есть некая среда JDeveloper от Oracle, но её SQLJ привязан к их базе данных и их пропиетарной Java-библиотеке. Но, даже если бы плагины для SQLJ для популярных сред были разработаны

4 Основная часть

Вообще абзац.

5 Список литературы

Ёж.