Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Межъязыковое взаимодействие OCaml и C++ со статическим контролем типов

Дипломная работа студента 544 группы
Дубчука Николая Павловича

Научный руководитель	.....	Я.А. Кириленко
ст. преподаватель	/ подпись /	
Рецензент	.....	В.С. Полозов
ст. преподаватель	/ подпись /	
“Допустить к защите”		
заведующий кафедрой,	.....	А.Н. Терехов
д.ф.-м.н., профессор	/ подпись /	

Санкт-Петербург
2007

Содержание

1 Введение

При решении задач автоматизированного реинжиниринга программ [1] неотъемлемым этапом является трансляция исходного текста в некоторое промежуточное представление, так называемое дерево разбора. При этом чаще всего используются средства автоматизации [2], которые значительно упрощают реализацию данной трансляции. Задачи реинжиниринга выдвигают к этим инструментам особые требования [3], а трансляция устаревших языков трудна даже с использованием современных генераторов, которые предоставляют недостаточно гибкие и удобные средства как для задания грамматик этих языков, так и для работы с неоднозначными грамматиками.

В дипломной работе Чемоданова И.С. „Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ“ [4] предлагается инструмент YARD. Этот генератор обладает выразительным и удобным входным языком и в большей степени отвечает требованиям автоматизированного реинжиниринга программ по сравнению с существующими разработками.

Применение генератора YARD затруднено в большинстве коммерческих продуктов, поскольку целевым языком инструмента является OCaml [5]. При использовании генератора в системе автоматизированного реинжиниринга приложений Modernization Workbench [6], реализованной на C++ [7], появляется проблема межъязыкового взаимодействия. Полученный при трансляции результат отображается в памяти как некоторое представление алгебраического OCaml-типа, а последующие этапы, извлекающие различную информацию [8], предполагают работу с типами данных языка C++. Для того, чтобы решить эту проблему, необходимо преобразовать структуры данных.

В результате проведенного обзора было выявлено, что существующие средства взаимодействия OCaml и C++ не обеспечивают в полной мере сохранения отношения над типами.

В дипломной работе рассматривается взаимодействие между языками программирования C++ и OCaml, и предлагается средство со статическим контролем типов для использования OCaml из C++. Статический контроль типов позволяет избежать ситуации, когда в OCaml-программе меняется тип, а в приложении на C++ данное изменение не учитывается. В таком случае вместо аварийного завершения программы во время исполнения, ошибка выявляется на стадии компиляции программы. Также контроль сохраняет равенство в C++ типов, равных в OCaml.

Структура дипломной работы В части 2 проводится обзор инструментов взаимодействия языков OCaml и C++. В части 3 предлагается отображение алгебраических типов OCaml в структуры языка C++. В части 4 раскрываются детали реализации предлагаемого в работе средства. В части 5 формулируются результаты работы.

2 Обзор инструментов взаимодействия языков OCaml и C++

Результат работы синтаксического анализатора, полученного при помощи генератора YARD — это некоторое представление в памяти алгебраического типа данных языка OCaml. Этот результат можно использовать только в приложениях на OCaml, если не производить некоторых действий для обеспечения совместимости с системой типов языка C++.

Для взаимодействия языков OCaml и C++ были изучены следующие средства:

- CamlIDL [14] (Caml Interface Description Language) — CamlIDL автоматизирует большинство рутинных и трудоемких задач взаимодействия OCaml-программ с библиотеками C.

  CamlIDL предоставляет возможность для использования COM-компонент (написанных на C/C++) из OCaml-программ, а также упаковывать OCaml-объекты в COM-компоненты, с которыми потом можно работать в C++. В качестве недостатков этого средства отметим, что использование технологии COM требует специальных знаний, и CamlIDL обеспечивает только динамический контроль типов, а не статический.

- SWIG [15] (Simplified Wrapper and Interface Generator) — инструмент, способствующий взаимодействию программ, написанных на C/C++, с большинством высокоуровневых языков программирования. Среди них есть скриптовые: Perl, PHP, Python, Tcl и Ruby, а также C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), Java, Modula-3 и OCaml.

  Основная цель этого средства —– реализовать взаимодействие с минимальными усилиями: в файлы заголовка программы добавляются специальный указания, по которым SWIG генерирует исходный код для склеивания C/C++ и нужного языка. Результат склеивания представляется в виде исполняемого файла исходной программы со встроенным интерпретатором этого языка.

Возможно, такое малое количество инструментов взаимодействия языков C++ и OCaml связано с небольшой популярностью последнего из них. Так, по оценкам компании TIOBE[1] языки C++ и OCaml занимают 3 и 48 место соответственно.

Рассмотренные средства не обеспечивают в полной мере необходимое для нас взаимодействие языков, не обладают статической проверкой типов, и поэтому было решено реализовать свой инструмент.

Были исследованы существующие подходы для обеспечения взаимодействия программ, написанных на разных языках программирования. Первая идея основана на использовании промежуточного файла (например, формата xml), в который сохраняется полученный результат, а после он особым образом обрабатывается программой на C++ и преобразуется в необходимые структуры данных. Это медленный способ, так как требуются большие временные затраты на запись

[1] http://www.tiobe.com/index.htm?tiobe_index

и чтение файла при каждом использовании OCaml-программы из приложения на C++. Альтернативный вариант — использование механизма межъязыкового взаимодействия и предоставление некоторого интерфейса, который автоматически преобразует OCaml-типы в интересующие нас структуры данных.

Стоит отметить, что межъязыковое взаимодействие позволяет совместно использовать модули и не задумываться, на каком языке они реализованы. Также оно компенсирует слабые стороны одного языка сильными другого и увеличивает функциональные возможности самого приложения.

Было рассмотрено множество средств и технологий [9], обеспечивающих межъязыковое взаимодействие между компонентами программного продукта. Отметим несколько технологий, в которых решается задача, похожая на нашу:

- Common Object Request Broker Architecture [10] (CORBA) — стандарт, разработанный консорциумом OMG (Object Management Group). CORBA позволяет совместно работать компонентам независимо от того, написаны ли они на разных языках или выполняются на разных машинах.

- Remote Procedure Call [11] (RPC) —- технология, позволяющая программам вызывать функции или процедуры в другом адресном пространстве (как правило, на удаленных машинах). В RPC решается ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках. Основными технологиями, обеспечивающими RPC являются: Sun RPC (RFC 1831), .Net Remoting, XML RPC, Java RMI.

- Component Object Model [12] (COM) —- это технологический стандарт от компании Microsoft, предназначенный для создания программного обеспечения на основе взаимодействующих распределенных компонентов, каждый из которых может использоваться во многих программах одновременно. Технология воплощает в себе идеи полиморфизма и инкапсуляции объектно-ориентированного программирования. Технология COM является универсальной и платформо-независимой, но закрепилась в основном на операционных системах семейства Windows. Основным понятием, которым оперирует технология COM, является COM-компонент. Программы, построенные на этой технологии, фактически не являются автономными программами, а представляют собой набор взаимодействующих между собой COM-компонентов. Каждый компонент имеет уникальный идентификатор (GUID) и может одновременно использоваться многими программами. Компонент взаимодействует с другими программами через COM-интерфейсы — наборы абстрактных функций и свойств.

- .NET [13] —- межплатформенная программная технология, предложенная фирмой Microsoft. Одной из основных идей .NET является совместимость компонент, реализованных на разных языках программирования платформы. Например, класс, написанный на C++ для .NET, может обратиться к методу

класса из библиотеки, реализованной на J#; на C# можно создать класс, унаследованный от класса, реализованного на Visual Basic .NET, а исключение, брошенное (throw) методом, написанным на C#, может быть перехвачено и обработано в J#. Такой подход реализуется благодаря общей системе типов (CTS), поддерживаемой всеми языками семейства. В нашем случае такой общей системы нет, и мы должны сами решить проблему передачи данных из OCaml-программы приложению на C++.

В конечном итоге было решено реализовать свой инструмент, который выполняет трансляцию алгебраических типов OCaml в структуры данных языка C++. Это средство должно полностью отвечать поставленной задаче (см. часть 1), а также сохранять равенство в C++ типов, эквивалентных в OCaml, которую можно обеспечить при помощи статического контроля типов.

3 Отображение алгебраических типов OCaml в структуры языка C++

В приложении на C++ результат работы OCaml-программы представляется в памяти некоторым значением типа value. Тем самым, теряется информация о типе этого результата, и ее необходимо извлекать из исходного текста OCaml-программы. При трансляции типов в соответствующие структуры языка C++ хочется сохранить эквивалентность типов, равных в OCaml. Сложность такой трансляции заключается в том, что в C++ нет базовых аналогов прямого произведения и прямой суммы языка OCaml, а также эти языки обладают разными представлениями о равенстве типов.

3.1 Аналог алгебраического типа в языке C++

Для прямой суммы предлагается использовать некоторый шаблонный класс variant, который предоставляет простое решение для манипулирования в одинаковом стиле наборами неоднородных данных.

Для прямого произведения применяется класс tuple — это коллекция фиксированного размера элементов, которые могут быть разных типов. Этот класс реализован посредством шаблонов.

Согласно стандарту языка C++ (раздел 9.2) нельзя описать структуру через самое себя, то есть внутри структуры `A` завести поле типа `A`. Рекурсивные структуры определяются только через указатели или ссылки. Для устранения циклической зависимости, присущей рекурсивным типам, используется класс recursive_wrapper.

Подробное описание классов приводится в части 4.2.

3.2 Равенство типов и соответствующих им структур

Как уже отмечалось, языки OCaml и C++ обладают разными представлениями о равенстве типов. Так, типы

```
type type1 = int * string
type type2 = int * string
```

являются синонимами одного и того же типа и равны в OCaml.

Если же объявить на C++ типы `type1` и `type2` как структуры с одинаковой реализацией, но разными именами, то присваивание одного объекта другому приведет к ошибке компиляции:

```
struct type1
{
  // реализация структуры type1
}

struct type2
{
```

```
  // такая же реализация, как и у type1
}

int main()
{
  type1 t1;
  type2 t2 = t1; // ошибка компиляции
}
```

Согласно стандарту языка C++ (раздел 7.1.3) при помощи typedef можно задать синоним типа, и при таком объявлении не создается новый тип. Два типа языка C++ будут равными, если это typedef на один и тот же тип. Согласно стандарту языка C++ (раздел 14.4) два шаблонных типа равны, если их имена совпадают, и их параметры одинаковых типов.

Тем самым, мы сохраним равенство в C++ для типов `type1` и `type2`, если объявим их следующим образом

```
typedef type_wrapper<boost::tuple<int, string> > type1;
typedef type_wrapper<boost::tuple<int, string> > type2;
type1 t1;
type2 t2 = t1;
      t1 = t2;
```

В этом примере используется шаблонная структура `type_wrapper` для того, чтобы обеспечить общий вид сгенерированных структур.

Для параметризованных типов, равных на OCaml

```
type 'a type3 = 'a * string
type 'a type4 = 'a * string
```

применение typedef, такое же, как для непараметризованных, приведет к ошибке компиляции:

```
template <typename_a>
typedef type_wrapper<boost::tuple<_a, string> > type3; // ошибка компиляции
```

Приходится „прятать“ typedef внутрь структуры:

```
template <typename _a>
struct type3
{
  typedef type_wrapper<boost::tuple<_a, string> > type;
}
```

Тем самым, мы сохранили эквивалентность типов `type3` и `type4` на C++:

```
type3<int>::type t3 = t2;
                 t2 = t3; // type3<int>::type эквивалентен type2
type4<int>::type t4 = t3;
                 t3 = t4; // type4<int>::type эквивалентен type3<int>::type
```

Для прямой суммы не приходится говорить о равенстве, так как она задается через теги, имена которых должны быть уникальными.

Таким образом, мы показали как сохранить в C++ эквивалентность типов, равных в OCaml .

4 Реализация

Механизм межъязыкового взаимодействия, реализованный в данной работе, состоит из нескольких этапов:

- Синтаксический разбор и анализ описаний типов интерфейса модуля OCaml-программы
- Генерация соответствующих структур языка C++ для каждого из типов (обеспечивается статический контроль типов)
- Передача управления приложением на C++ программе на OCaml
- Трансляция результата работы OCaml-программы в соответствующую структуру языка C++

В следующих частях проводится подробное описание каждого из этапов.

4.1 Синтаксический разбор и анализ описаний типов интерфейса модуля

Для извлечения информации о типах из исходного файла программы на OCaml необходимо получить интерфейс модуля (при помощи компилятора `ocamlc`). Далее необходимо провести синтаксический анализ описаний типов этого интерфейса модуля и сгенерировать соответствующие структуры на C++ для каждого из типов.

Для синтаксического разбора интерфейса можно создать парсер вручную или использовать средство (генератор или принтер). При реализации разбора был сильно модифицирован стандартный принтер `pr_o` на Camlp4 [16], который для файла производит синтаксический анализ и печатает его содержимое. В новом принтере собирается информация о каждом из типов: параметры типа, названия тегов и типы каждого из них. Таким образом, информацию о типах из файла мы уже можем извлечь, и теперь необходимо сгенерировать соответствующие им структуры языка C++.

4.2 Генерация соответствующих структур языка C++ для каждого алгебраического OCaml-типа

Необходимым требованием к этой генерации выдвигается сохранение равенства типов, равных на OCaml. Как обеспечить эту эквивалентность, рассказывается в части 3.2.

Для реализации данной трансляции понадобилась библиотека структур данных из Boost, включающая в себя `recursive_wrapper`, `tuple` и `variant`.

Boost [17] — это коллекция библиотек, расширяющих C++. Они свободно распространяются по лицензии Boost Software License вместе с исходными кодами. Проект был начат членами комитета по стандартизации C++ (Standards Committee Library Working Group) после принятия стандарта C++, когда в него не включили некоторые библиотеки. Сейчас его разрабатывают тысячи программистов.

В Boost широко применяются метапрограммирование и обобщенное программирование с активным использованием шаблонов.

Разработчиками отмечается, что уже десять Boost библиотек включено в C++ Standards Committee's Library Technical Report (TR1), чтобы в будущем стать частью стандарта C++.

Boost содержит множество библиотек для C++, которые предназначены для широкого спектра задач. Например, в Boost есть библиотеки алгоритмов над графами, работы со строками и текстом, шаблонного метапрограммирования и другие.

Рассмотрим процесс трансляции значений каждого OCaml-типа.

Трансляция значений примитивного типа языка OCaml Примитивные типы языка OCaml (int, float, bool, string, char) можно преобразовать в соответствующие им типы языка C++ при помощи средств (макросов), предоставляемых разработчиками OCaml.

Часть макросов, понадобившаяся при реализации, представлена в таблице 1.

Макрос	*Описание*
Int_val(v)	преобразует `v` в значение типа `int` языка C++
Long_val(v)	преобразует `v` в значение типа `long` языка C++
Bool_val(v)	преобразует `v` в значение типа `bool` языка C++
Double_val(v)	преобразует `v` в значение типа `double` языка C++
String_val(v)	преобразует `v` в значение типа `char *` языка C++

Таблица 1: Макросы преобразования, предоставляемые разработчиками.

Стоит отметить, что в OCaml есть примитивный тип unit, аналогом которого может служить структура из пространства имен `caml2cxx` (см. часть 4.6).

Трансляция значений прямой суммы Для прямой суммы

```
type t  = A of int | B of long
```

предлагается использовать структуру, хранящую информацию о тегах и поле типа `boost::variant<int, long>` — шаблонный класс, который предоставляет простое решение для манипулирования в одинаковом стиле наборами неоднородных данных.

Приведем пример записи и извлечения значения для объекта класса `variant`

```
boost::variant<int, long> data;
int a = 1;
data = a; // запись числа
a = boost::get<int>(data); // извлечение числа
```

Запись осуществляется при помощи обычного присваивания, а для извлечения информации необходимо использовать функцию `boost::get`.

Трансляция значений прямого произведения Прямое произведение или тип tuple — это коллекция фиксированного размера элементов, которые могут быть разных типов. OCaml имеет встроенную поддержку для tuple, в отличие от языка C++. Библиотека Boost.Tuple содержит реализацию типа tuple посредством шаблонов.

Для прямого произведения типов `int`, `bool` и `t` (`int * bool * t`) аналогом на C++ будет `boost::tuple<int, bool, t>`.

Кортеж получается конкретизацией шаблона tuple. Параметры этого шаблона задают типы хранимых элементов. Конструктор tuple получает элементы кортежа как аргументы. Для `n`-элементных кортежей можно вызывать конструктор с числом аргументов k, где $0 \leq k \leq n$. К примеру:

```
tuple<int, bool, t>()
tuple<int, bool, t>(1)
tuple<int, bool, t>(1, true)
```

Если для элемента не задано начального значения, то он инициализируется по умолчанию (и, следовательно, должен допускать такую инициализацию). Кортежи могут быть созданы с помощью вспомогательной функции make_tuple. Это делает создание кортежей более удобным, избавляя программиста от необходимости явно объявлять типы элементов. Доступ к элементам кортежа осуществляется при помощи выражения `t2.get<N>()` или `get<N>(t2)`, где N — целочисленное константное выражение, задающее индекс элемента в кортеже. В зависимости от того, квалифицирован ли объект `t2` как константный или нет, `get` возвращает N-й элемент как ссылку на константный или неконстантный тип. Индекс первого элемента 0, поэтому N должен быть в диапазоне от 0 до $k-1$, где k — число элементов в кортеже. Нарушения этих ограничений обнаруживаются во время компиляции.

Трансляция параметризованного типа Для параметризованного типа

```
type ('a, 'b) t3 = 'a * int * 'b
```

будет сгенерирована шаблонная структура, использующая `boost::tuple`

```
template <typename _a, typename _b>
struct t3
{
  typedef boost::tuple<_a, int, _b> type;
};
```

Стоит уделить некоторое внимание проблеме именования параметров полученной шаблонной структуры, так как имя одного из параметров может совпадать с именем самой структуры. В OCaml такой проблемы не стоит, поскольку имена параметров всегда начинаются с соответствующего символа, который не может использоваться в качестве префикса для имени типа. В свою очередь, идентификатор языка C++ не может начинаться с этого символа, поэтому было решено префиксом параметров использовать „_“, а в случае совпадения имен добавлять любой символ.

Трансляция значения рекурсивного типа Как уже отмечалось, в C++ нельзя описать структуру через самое себя, необходимо использовать указатели или ссылки.

Аналогом рекурсивного типа

```
type t4 = E of t4
```

на C++ будет `boost::recursive_wrapper<t4>`, который разрешает циклическую зависимость, присущую рекурсивным типам.

4.2.1 Общий вид сгенерированной структуры

Для некоторого алгебраического типа `t` с прямой суммой

```
type t = A of тип_A | ...
```

опишем общий вид соответствующей структуры на C++ :

```
struct t
{
  struct type
  {
    // перечисление используемых тегов
    enum tags { A = 0, ... };

    // методы определения тега, который хранится в поле data
    bool isA() const { return tag == A; }
    ...

    // методы доступа
    тип_A getA() const { return boost::get< тип_A >(data); }
    void setA(тип_A a) { data = a; tag = A; }
    ...

    // конструктор от value
    t (value v) : data(fill(v)) {}

  private:
```

```
    tags tag;  // имя тега
    boost::variant< тип_A, ... > data; // данные

    // методы преобразования каждого из тегов
    тип_A getValA(value v)
    {
      return (valTo <тип_A> (Field(v, 0)));
    }
    ...

    boost::variant< тип_A, ... >  fill (value v)
    {
      switch (Tag_val(v))
      {
        case 0:
          tag = A;
          return getValA(v);
        ...

        default: // ошибка в номере тега
        assert(!"Invalid tag for type t");
      }
    }
  }
};
```

В данной структуре хранится поле `data` типа `boost::variant`. Также имеется конструктор от `value`, в котором происходит преобразование из OCaml-типа в соответствующую структуру на C++ при помощи методов преобразования каждого из тегов. У структуры имеются методы доступа и определения тега.

Стоит отметить, что сгенерированные структуры обладают читабельным кодом, оператором вывода, и каждая из них инкапсулирует трансляцию значения value внутри своего конструктора.

4.3 Передача управления приложением на C++ программе на OCaml

Рассмотрим в качестве примера взаимодействия OCaml и C++ (использование OCaml-функции в приложении на C++) следующую задачу. Допустим, у нас имеется функция `parseFile`, реализованная на OCaml , которая в качестве параметра получает имя файла, производит трансляцию исходного текста в некоторое промежуточное представление (дерево разбора), а потом возвращает его в качестве результата своей работы. И мы хотим из приложении на C++ запустить эту функцию и преобразовать полученное дерево разбора в структуру языка C++ .

Существует два способа передачи результата работы OCaml-функции в приложение на C++. Первый из них предполагает вызов из OCaml-программы функции на C++, а второй — запуск из C++ функции на OCaml. Второй способ больше подходит для нашей задачи, так как вся реализация межмодульного взаимодействия будет располагаться только в приложении на C++. Этот способ использует механизм обратного вызова (callback), позволяющий зарегистрировать функции в OCaml, которые мы хотим использовать в C++. Вся реализация со стороны OCaml требует добавление всего одной строчки кода

```
let _ =  Callback.register "parse" parseFile
```

В этом примере регистрируется функция `parseFile`, которая из C++ будет вызываться как `parse`.

Приложение на C++, вызывающее эту функцию и преобразующее полученный результат ее работы, имеет вид

```
extern "C"
{
  #include <caml/alloc.h>
  #include <caml/callback.h>
}

value start()
{
  static value * parse_closure = NULL;
  if (parse_closure == NULL)
    parse_closure = caml_named_value("parse");

  assert(parse_closure);

  return callback(*parse_closure, copy_string("filename"));
}

CTree convert(value ocamlTree)
{
  CTree tree;
  // преобразование ocamlTree в tree
  return tree;
}

int main(int args, char** argv)
{
  caml_startup(argv);

  value camlTree = start();
```

```
  CTree tree = convert(camlTree);

  // дальнейшая работа с преобразованным деревом tree

  return 0;
}
```

В этом примере вначале вызывается функция `caml_startup`. Далее из функции `start` запускается `caml_named_value`, которая возвращает замыкание зарегистрированной в OCaml функции, которая может вызываться из нашего приложения несколько раз. Для того, чтобы каждый раз не запускать `caml_named_value`, используется статический указатель на `value` — `parse_closure`.

Следующим шагом мы хотим преобразовать полученное от OCaml-функции дерево разбора в соответствующую структуру языка C++ — `CTree`. Для этого используется функция `convert`, принимающая `ocamlTree` типа `value` и возвращающая соответствующее значение структуры языка C++ — `tree`.

При помощи функции `callback` мы вызываем функцию из OCaml , передав ей имя интересующего нас файла. Функция `callback` возвращает значение типа `value`. Если функции на OCaml потребуется передать несколько параметров, а не один, как в этом примере, то это можно сделать следующим образом:

```
callback2(f, v1, v2)     - вызов f с двумя параметрами
callback3(f, v1, v2, v3) - вызов f с тремя параметрами
callbackN(f, n, arr)     - вызов f с n параметрами,
                           хранящимися внутри массива arr
```

Таким образом, при помощи механизма обратного вызова мы регистрируем OCaml-функцию, которую потом вызываем из приложения и транслируем результат работы в структуру на C++ .

В частях 4.2 и 4.3 рассказывается о том, как извлечь информацию о типах, определенных в OCaml-программе, и какие структуры языка C++ им соответствуют. А в части 4.4 описывается реализация функции `convert`.

4.4 Трансляция результата, полученного из OCaml

Для того, чтобы преобразовать полученный из OCaml-программы результат (`camlTree`) в интересующую нас структуру на C++ , необходимо вызвать ее конструктор от этого значения. В конечном итоге функция `convert` принимает вид:

```
Tree convert(const value & v)
{
  return Tree(v);
}
```

Стоит отметить, что такое преобразование имеет место для всех сгенерированных структур, так как те, в свою очередь, располагают соответствующим конструктором от типа `value`. Такой подход обеспечивает универсальность трансляции OCaml-значения в соответствующую структуру и инкапсулирует само преобразование.

Раскроем детали преобразования. В случае алгебраического типа, являющегося примитивным или прямым произведением примитивных, для C++ используется `type_wrapper`, причем сохраняется равенство типов в C++ .

Сложности появляются для прямой суммы, имеющей несколько тегов. Например, для типа

```
type bin_tree = Leaf of int | Tree of bin_tree * bin_tree
```

при трансляции в интересующую нас структуру на C++ (`BinTree`) придется понять, с каким из тегов мы имеем дело. Определять это по размерности каждого из тега (у `Leaf` — 1, у `Tree` — 2) неправильно, так как для двух тегов с одинаковой размерностью такое решение будет неверным. Благодаря макросам `Tag_val` и `Field(v, n)` можно узнать номер тега и извлечь `n`-ое поле типа `v`. Таким образом, решение примет вид:

```
BinTree convert(value camlTree)
{
  BinTree cTree;
  switch (Tag_val(v))
  {
    case 0: // разбираем случай Leaf
      ...
      break;
    case 1: // разбираем случай Tree
      ...
      break;
    default: // ошибка в номере тега
      assert(!"Invalid tag");
  }
  return cTree;
}
```

4.5 Использование YARD в приложениях на C++

С точки зрения реализации все выполнено для того, чтобы использовать инструмент YARD в приложениях на C++ .

В качестве примера при помощи YARD был сгенерирован калькулятор — синтаксический анализатор, вычисляющий алгебраическое выражение. Калькулятор возвращал результат вычисления выражения, переданного ему из программы на C++ .

При интеграции инструмента YARD в систему автоматизированного реинжиниринга программ Modernization Workbench может возникнуть проблема

совместимости сгенерированных структур с внутренним представлением данных системы. Поэтому для интеграции генератора необходимо будет выполнить трансляцию структур в это представление.

4.6 Пространство имен caml2cxx

Для сгенерированных структур на C++ предоставляется пространство имен `caml2cxx`, в котором определены вспомогательные функции преобразования, шаблонная структура `type_wrapper` и структура `unit`:

```
namespace caml2cxx
{
  // шаблонная функция преобразования OCaml-значений
  // в соответствующие им структуры на C++
  template <typename T>
  T valTo(value val)
  {
    // преобразование хранящегося внутри другого
    // значения непримитивного типа
    return T(val);
  }

  // частичное применение функции valTo к типу int
  template <>
  int valTo(value val)
  {
    return Int_val(val);
  }
  ...
  // частичные применения функции valTo к остальным
  // примитивным типам (string, bool, float ...)
  ...

  // шаблонная структура-оболочка
  // для примитивных OCaml-типов и их прямого произведения
  template <typename T> struct type_wrapper
  {
    T getData() const { return data; }
    void setData(T a) { data = a; }

    explicit type_wrapper (value v)
    {
      setData(valTo <T> (Field(v, 0)));
    }
```

```
    private:
      T data;
  };

  // структура, соответствующая примитивному OCaml-типу unit
  struct unit
  {
  };
}
```

В caml2cxx содержится `valTo` — шаблонная функция преобразования OCaml-типов в структуры языка C++. Для всех примитивных типов определяются частичные применения этой функции, реализации которых используют соответствующие макросы (см. часть 4.2). Для непримитивных типов предполагается, что уже сгенерированы структуры на C++, которые могут конструироваться по типу `value`.

Также в пространстве имен реализуется шаблонная структура-оболочка для примитивных OCaml-типов и их прямого произведения, которая обеспечивает похожесть всех сгенерированных структур.

Аналогом примитивного OCaml-типа unit в C++ может служить структура `unit`, определенная в этом пространстве имен.

5 Результаты

В ходе данной дипломной работы были достигнуты следующие результаты:

- Выполнен обзор существующих средств взаимодействия OCaml и C++
- Предложено отображение из алгебраических типов OCaml в структуры языка C++ с использованием библиотеки Boost
- Реализован инструмент, выполняющий трансляцию структур данных из OCaml в C++
- Реализовано межъязыковое взаимодействие программы-калькулятора, сгенерированной YARD-ом, с приложением на C++

Список литературы

[1] *Эрлих Л.А.* Модернизация старых программ как ключевой фактор электронной коммерции // Автоматизированный реинжиниринг программ / Под ред. проф. А.Н. Терехова и А.А. Терехова. — СПб.: Издательство С.-Петербургского университета, 2000 — с. 20-42.

[2] *Дубчук Н.П., Чемоданов И.С.* Обзор современных средств автоматизации создания синтаксических анализаторов // Системное программирование. — СПб.: Изд-во С.-Петерб. ун-та, 2006. — с. 286–316.

[3] *Mark G. J. van den Brand, A. Sellink, C. Verhoef* Current Parsing Techniques in Software Renovation Considered Harmful // IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension. — IEEE Computer Society, Washington, 1998.

[4] *Чемоданов И.С.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ: Дипломная работа С.-Петербург. гос. ун-т, мат.-мех. ф-т. СПб., 2007.

[5] http://caml.inria.fr (сайт разработчиков языка программирования OCaml)

[6] Relativity Modernization Workbench User Manual. Available with the Modernization Workbench system ©Relativity Technologies (www.relativity.com).

[7] ISO/IEC International Standard 14882, Programming Languages — C++ (стандарт языка программирования C++)

[8] *Бульонков М.А., Бабурин Д.Е.* HyperCode — открытая система визуализации программ // Автоматизированный реинжиниринг программ / Под ред. проф. А.Н. Терехова и А.А. Терехова. — СПб.: Издательство С.-Петербургского университета, 2000 — с. 165-183.

[9] http://rsdn.ru/archive/vc/issues/pvc036.htm (обзор технологий межъязыкового взаимодействия)

[10] http://www.corba.org/ (сайт разработчиков технологии CORBA)

[11] http://www.cs.cf.ac.uk/Dave/C/node33.html (сайт разработчиков технологии RPC)

[12] http://www.microsoft.com/com/default.mspx (сайт разработчиков технологии COM)

[13] http://www.microsoft.com/net/ (сайт разработчиков технологии .NET)

[14] http://caml.inria.fr/pub/old_caml_site/camlidl (сайт разработчиков CamlIDL)

[15] http://www.swig.org (сайт разработчиков SWIG)

[16] http://caml.inria.fr/pub/old_caml_site/camlp4/index.html (сайт разработчиков Camlp4)

[17] http://www.boost.org (сайт разработчиков библиотеки Boost)

[18] http://www.boost.org/doc/html/variant.html (класс variant из библиотеки Boost)

[19] http://www.boost.org/libs/tuple/doc/tuple_users_guide.html (класс tuple из библиотеки Boost)

[20] http://www.boost.org/doc/html/boost/recursive_wrapper.html (класс recursive_wrapper из библиотеки Boost)