Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Факультет математики, механики и компьютерных наук

Направление подготовки 010400 — «Информационные технологии»

АВТОМАТИЧЕСКОЕ ПОСТРОЕНИЕ ОГРАНИЧЕНИЙ В МОДЕЛЬНОМ ЯЗЫКЕ ПРОГРАММИРОВАНИЯ С ШАБЛОНАМИ ФУНКЦИЙ И АВТОВЫВОДОМ ТИПОВ

Выпускная квалификационная работа
на степень бакалавра

студентки
Ю. В. Беляковой

Научный руководитель:
доцент, кандидат физ.-мат. наук
С. С. Михалкович

Ростов-на-Дону
2012

Оглавление

Введение

Одним из популярных направлений в области современного программирования является обобщенное программирование. **Обобщенное программирование (generic programming)** — парадигма программирования, предусматривающая написание универсального кода пригодного для повторного использования. Пионерами в этой области считаются А. Степанов и Д. Мёссер [1]. Наиболее известный пример обобщенной библиотеки — это стандартная библиотека C++ (Standard Template Library, STL) [2].

Подходы к обобщенному программированию

Обобщенное программирование в стандартной библиотеке C++ реализуется посредством **шаблонов (templates)**. Шаблоны позволяют описывать алгоритмы и структуры данных, в которых кроме значений *конкретных типов* можно также использовать значения *типов-параметров шаблона*. Чтобы использовать шаблон, его нужно **инстанцировать**, то есть подставить вместо параметров шаблона реальные типы. Основная особенность шаблонов C++ заключается в том, что код шаблона подвергается проверке синтаксиса и только *минимальной* проверке семантики. Полный анализ проводится лишь при инстанцировании шаблона. Таким образом можно написать код, который не компилируется ни при каких значениях параметров шаблона. Зато в теле шаблона можно использовать практически все возможности языка.

Другая концепция принята для **универсальных шаблонов .NET**

(generics) [3]. В отличие от шаблонов C++ для значений типов-параметров шаблона можно использовать только явно разрешенные операции: это возможности базового класса всех типов `Object` и возможности, явно указанные в заголовке в виде ограничений на параметры шаблона. Ограничения могут быть следующих видов: реализация интерфейса; определенный базовый класс, который имеет конструктор по умолчанию; ссылочный тип или тип значений. Такие ограничения значительно сужают возможности универсальных шаблонов, при этом требуя от пользователя прописывать их явно. Но проверка корректности инстанций сводится к простой проверке ограничений.

В *теории типов* поведение обобщенного кода называют **параметрическим полиморфизмом (parametric polymorphism)** [4], а сам код — **полиморфным**. В функциональном языке Haskell параметрический полиморфизм реализуется средствами **классов типов** [5] (в функциях можно использовать значения не только конкретных типов, но и *типовых переменных*-экземпляров классов типов). Классы типов похожи на универсальные шаблоны .NET: для значения типа-экземпляра некоторого класса доступны функции данного класса. Тип может реализовывать различные классы. Однако в отличие от шаблонов .NET есть возможность определять классы, накладывающие ограничения сразу на несколько типов, — *мультипараметрические* классы типов. С их помощью можно, к примеру, создать аналог перегруженной функции с несколькими аргументами.

Примечание. Перегрузка (соответствие одного и того же символа функции разным реализациям) в теории типов носит название **специализированного полиморфизма (ad-hoc polymorphism)**.

Преимущества и недостатки

Если остановиться на реализации шаблонов (в C++ и .NET), то коротко эти подходы можно охарактеризовать так:

1. «Можно все», минимальная проверка обобщенного кода, полная проверка кода для каждой инстанции.
2. «Можно только то, что разрешено», полная проверка обобщенного кода, проверка ограничений для инстанции.

Преимуществом первого подхода являются большие возможности в написании обобщенного кода, недостатки заключаются в позднем обнаружении ошибок и необходимости повторной проверки кода при инстанцировании. Преимущества второго — раннее обнаружение ошибок и простая проверка ограничений при инстанцировании. Недостаток — ограниченные возможности шаблонов.

Замечание. В стандарт C++0x планировалось включение **концептов** [6]. Концепты являются примером второго подхода к шаблонам (с ограничениями), но по сравнению с универсальными шаблонами .NET предоставляют гораздо больше возможностей (к примеру, они позволяют накладывать ограничения, устанавливающие связь между несколькими типами). Пока включение концептов отложено до будущего стандарта.

Цель данной работы — исследовать альтернативный механизм шаблонов, совмещающий преимущества обоих подходов. На примере простого модельного языка программирования будет рассмотрен такой механизм шаблонов функций, при котором:

1. на этапе компиляции шаблона будет проведена *максимальная проверка семантики* и *автоматически собраны ограничения* на параметры шаблона;
2. на этапе инстанцирования потребуется проверить *соответствие типов ограничениям*.

Резюмируя вышесказанное, ещё раз обозначим достоинства подхода, основанного на сборе ограничений:

1. раннее обнаружение ошибок (на этапе компиляции шаблона проводится не только синтаксический, не и семантический анализ: если шаблон откомпилирован, значит существуют корректные инстанции, иначе будет обнаружена ошибка);
2. упрощается этап инстанцирования шаблона — достаточно проверить ограничения;
3. информацию, полученную в результате компиляции шаблона, можно использовать для помощи Intellisense.

Постановка задачи

1. Разработать простой модельный императивный язык программирования с несколькими базовыми типами и поддержкой шаблонов функций. С этой целью разработать грамматику языка.
2. Создать front-end часть компилятора, используя средства автоматической генерации лексических и синтаксических анализаторов.
3. Разработать алгоритмы анализа шаблонов функций, построения ограничений на шаблоны, анализа ограничений, инстанцирования шаблонов.
4. Реализовать middle-end часть компилятора, задача которой — полный семантический анализ синтаксически корректной программы, включая применение вышеуказанных алгоритмов.

Глава 1

Модельный язык PollyTL

Поскольку целью работы (см. стр. 6) является *исследование подхода к обобщенному программированию*, необходимо было выбрать некоторый язык программирования, поддерживающий написание и использование шаблонов, и написать front-end и middle-end части компилятора для данного языка. Реальные языки содержат множество различных возможностей и деталей, не имеющих прямого отношения к обобщенному программированию, а потому слишком громоздки для данной задачи.

Было принято решение разработать простой модельный язык с несколькими базовыми типами и поддержкой шаблонов функций. И уже на примере этого языка заниматься исследованием полиморфных (или обобщенных) функций — одного из аспектов обобщенного программирования.

Примечание. Как было отмечено ранее, обобщенное программирование включает в себя работу не только с алгоритмами, но и структурами данных, которые в настоящей работе не исследуются.

1.1 Основные характеристики

Язык PollyTL — это строго типизированный императивный язык программирования со статической типизацией. Система типов состоит из

нескольких базовых типов и *функциональных типов*, которые строятся из базовых. Создание пользовательских типов не предусмотрено.

Преобразования типов отсутствуют. Переменной может быть присвоено только выражение того же типа.

Кроме оператора присваивания и операций с базовыми типами есть условный и оператор цикла. Язык позволяет описывать и вызывать функции и шаблоны функций. Вложенное описание функций отсутствует. Шаблоны функций могут содержать вызовы других шаблонов (при этом шаблоны могут быть вызваны как с конкретными типами, так и типами-параметрами внешнего шаблона). Функции могут быть перегружены, шаблоны — нет.

PollyTL имеет ряд особенностей, нехарактерных для императивного языка:

1. функции являются значениями функционального типа `S` $\rightarrow$ `T`;
2. возможно частичное применение функции (то есть вызов функции от меньшего числа аргументов, при этом результатом является функция);
3. экземпляры перегруженной функции должны иметь различные типы, а потому могут отличаться только типом возвращаемого значения.

Такой «функциональный» подход добавляет гибкости в работе с функциями, но имеет и недостаток: тип выражения не всегда может быть определен. К примеру, если определены два экземпляра перегруженной функции `f` с типами `Bool` $\rightarrow$ `(Int` $\rightarrow$ `Int)` и `Bool` $\rightarrow$ `(Double` $\rightarrow$ `Double)`, то при вызове `f(true)` может оказаться, что невозможно определить, какой именно экземпляр `f` должен быть вызван.

1.2 Описание

1.2.1 Типы

Система типов состоит из именованных (включая три базовых) и функциональных типов.

Базовые типы:

- `Bool` — логический тип. Значениями являются константы `true` и `false`.
  Если `a, b :: Bool`[1], то следующие выражения допустимы и имеют тип `Bool`: `!a, a && b, a || b`.
- `Int` — целые числа, например: `1, -4, 567`.
  Если `a, b :: Int`, то следующие выражения допустимы и имеют тип `Int`: `-a, a + b, a - b, a * b, a` **div** `b, a` **mod** `b` (где **div**, **mod** — ключевые слова). Выражение `a / b` имеет тип `Double`.
- `Double` — числа с плавающей точкой, например: `-3.14159, 2.7`.
  Если `a, b :: Double`, то следующие выражения допустимы и имеют тип `Double`: `-a, a + b, a - b, a * b, a / b`.

Если `a, b` являются выражениями одного базового типа, то допустимы операторы сравнения на равенство и неравенство: `a == b, a != b :: Bool`. Для выражений числовых типов `a, b` $\in$ `{Int, Double}` можно использовать операторы отношения `a < b, a > b, a <= b, a >= b :: Bool`.

Функциональные типы. Это типы вида `S` $\rightarrow$ `T`, где `S, T` — это базовые или функциональные типы. Например:
`(Int` $\rightarrow$ `Int)` $\rightarrow$ `(Int` $\rightarrow$ `Bool)` $\rightarrow$ `Int` $\rightarrow$ `Int`.

Функциональные типы правоассоциативны, то есть запись `Int` $\rightarrow$ `Int` $\rightarrow$ `Int` эквивалентна записи `Int` $\rightarrow$ `(Int` $\rightarrow$ `Int)`.

[1] Здесь и далее двойное двоеточие в выражении `<expr :: type>` означает, что выражение `expr` имеет тип `type`

Выражением функционального типа может быть переменная (или формальный параметр функции), имя функции, а также результат вызова функции. Функцию или функциональную переменную можно вызвать (применить к функции аргументы) следующим образом:
`<имя функции>(<список фактических параметров>)`. Более подробно о синтаксисе функций будет сказано в п. 1.2.2.

В системе типов также неявно присутствует тип `void`. Он используется для обозначения типов функций без параметров (`void` $\rightarrow$ `T`) и процедур — функций без возвращаемого значения (`S` $\rightarrow$ `void`). Формально это тип, множество значений которого состоит из одного элемента. Описать значение типа `void` явно нельзя.

1.2.2 Синтаксис и семантика

Структура программы. Любая программа на языке PollyTL должна содержать функцию `main` и может содержать секцию описаний.

```
[<declarations>]
main
    [<statements>]
end

<declarations> ::= <declaration>
    | <declarations> <declaration>
<declaration> ::= <variable definition>
    | <function definition>
    | <template definition>
```

Declarations — секция описаний, statements — операторы.

Секция описаний может содержать описание переменных, функций и шаблонов функций. Секция операторов — оператор присваивания, описание переменных, вызов функции или шаблона функции, условный

оператор и оператор цикла (while).

Комментарии. Однострочные комментарии начинаются с символов //. Многострочные ограничиваются символами /* в начале и */ в конце. Многострочные комментарии могут быть вложенными.

Операторы. Переменные могут быть описаны двумя способами: с явным указанием типа и без. Следующим образом можно описать переменную с явным указанием типа:

```
<тип> <список переменных>;
<список переменных> ::= <описание переменной>
| <список переменных>, <описание переменной>
<описание переменной> ::= <имя переменной> [= <выражение>]
```

При определении переменной тип можно не указывать:

```
var <имя переменной> = <выражение>;
```

Далее приведен пример использования операторов присваивания и описания переменных.

```
main
    Int x, y = 3, z;      // описание переменных с присваиванием
                          // некоторых начальных значений
    x = 5 + y;            // оператор присваивания
    Bool b = false;
    var t = true || b;    // описание переменной с автоматическим
                          // выведением типа
end
```

При описании переменной через **var** тип переменной *выводится автоматически* по типу выражения справа. В операторе присваивания и описании переменной с начальным значением присваиваемое выражение должно иметь тот же тип, что и переменная.

Условный оператор имеет следующий синтаксис:

```
if <condition> then
    <statements>
{elif <condition> then
    <statements}
[else
    <statements>]
fi
```

В качестве условия (`<condition>`) может выступать только выражение типа `Bool`. `Elif-` и `else`-части оператора не являются обязательными. Пример:

```
if a < 0 then
    b = -b;
elif (a > 0) && (a < 10) then
    b = b * 2;
else
    b = 0;
fi
```

Так определяется **оператор цикла** (выражение `<condition>` должно иметь тип `Bool`):

```
while <condition> do
    <statements>
endw
```

Например:

```
var i = 0;
while i < 10 do
    i = i + 1;
    // do something
endw
```

Функции. Для определения функций используется следующий синтаксис:

```
[<result type>] fun (<formal parameters list>)
    <statements>
end

<formal parameters list> ::= <empty>
    | <formal parameters list>, <formal parameter>
<formal parameter> ::= <type> <parameter name>
```

Возвращаемый тип функции можно не указывать, тогда он будет выведен автоматически. Если возвращаемый тип отличен от `void`, то во всех ветках тела функции обязательно должен присутствовать оператор `return <expression>`:

```
// someSqr :: Int -> Int -> Int
fun someSqr(Int a, Int b)
    if a > b then
        var s = a + b;
        return s * s;
    else
        return 0;
    fi
end
```

Вызвать функция можно следующим образом:

```
<function name>([<factual parameters list>])
<factual parameters list> ::= <expressions list>
<expressions list> ::= <expression>
    | <expressions list>, <expression>
```

Функция может быть присвоена функциональной переменной. Для функции из примера выше допустимы следующие варианты использования:

```
Int r = someSqr(2, 3);
var f1 = someSqr(2);            // f :: Int -> Int
var f2 = someSqr;               // f :: Int -> Int -> Int
```

```
Int -> Int -> Int f3 = someSqr;
```

Если функция определена без параметров, то считается, что она имеет единственный параметр типа `void`, и тип всей функции — это `void` $\to$ `T`. Процедуру можно определить как функцию с типом `void` возвращаемого значения, то есть процедура — это функция типа `S` $\to$ `void`. Процедура может содержать пустые операторы `return;`, при этом их наличие во всех ветках не обязательно.

Замечание. Скобки при вызове функции обязательны.

Пример:

```
// voidFun :: void -> Int
fun voidFun()
    Int x;
    // do domething
    return x;
end
// p :: Int -> void
void fun p(Int x)
    Bool b, b1;
    // b = ...
    if b then
        // do domething
        return;
    elif b1 then
        // do domething
    fi
end

main
    var n = voidFun();          // n :: Int
    p(-256);
end
```

Замечание. «Непрерывные» вызовы вида `f(x)(y)` не предусмотрены. Чтобы совершить подобное использование функции, нужно присвоить `f(x)` некоторой переменной `g`, а затем сделать вызов `g(y)`. Например:

```
Int fun Sum(Int a, Int b)
    return a + b;
end
Int->Int fun alt(Bool b, Int -> Int f, Int -> Int g)
    if b then
        return f;
    else
        return g;
    fi
end

main
    // ERROR
    var z = alt(true, Sum(2), Sum(3))(24);
    // Success
    var f = alt(true, Sum(2), Sum(3));
    var z = f(24);
end
```

Рассмотрим пример использования перегруженных функций. В пункте 1.1 было отмечено, что разрешить вопрос о том, какой именно экземпляр функции нужно вызывать, можно не всегда.

```
// or :: Bool -> Bool -> Bool
fun or(Bool a, Bool b)
    return a || b;
end
// or :: Bool -> Bool -> Int
fun or(Bool a, Bool b)
    if a || b then
        return 1;
```

```
    else
        return 0;
    fi
end

main
    var x = 5;
    // <condition> :: Bool => or :: Bool -> Bool -> Bool
    if or(true, false) then
        x = -5;
    fi;
    // or :: Bool -> Bool -> Int
    Int n = or(false, false);
    // ERROR
    var u = or(true, true);
end
```

В данном случае при описании переменной с автовыводом типа мы не можем определить, какой именно `or` следует вызвать. А вот при использовании `or` в качестве условия известно, что условное выражение должно иметь тип `Bool`.

Замечание. Все операции над базовыми типами являются перегруженными функциями.

Шаблоны функций. Синтаксис шаблонов похож на синтаксис обычных функций, но после имени функций должен быть объявлен список параметров шаблона. В списке формальных параметров и теле шаблона могут быть использованы выражения типов-параметров шаблона.

```
T fun t_sum[!T](T x, T y)
    return x + y;
end

fun t_app[!F, T](F f, T x)
    return f(x);
```

```
end
```

При вызове шаблона параметры шаблона выводятся по аргументам вызова. Если не все параметры могут быть выведены, их нужно указать явно. Например, для шаблона функции (без параметров) `temp` с параметрами шаблона `T1, T2, T3` следующие вызовы равнозначны:

```
temp[!Bool, Int, Double]();
temp[!T1=Bool, Int, Double]();
temp[!Bool, T2=Int, Double]();
temp[!T1=Bool, T2=Int, T3=Double]();
temp[!T1=Bool, T3=Double, T2=Int]();
```

Более подробно шаблоны функций будут рассмотрены в главе 2.

1.3 Вывод информации о типах

Для наглядности и удобства работы с алгоритмами вывода типов и анализа шаблонов требуется простой способ узнать тип выражения. Это можно сделать, используя директиву компилятора `#print_type(<expressions list>)`. Она печатает типы выражений из списка `<expressions list>`. Эта директива может быть использована внутри тела функции (включая `main`) или шаблона.

```
fun sum(Int a, Int b)
    return a + b;
end
fun sum(Double a, Double b)
    return a + b;
end

main
    var x = 5 + 4.5*(-245);
    #print_type(x)
    #print_type(sum)
```

```
    #print_type(sum(1))
    #print_type(sum(x))
end
```

Результат:

```
x :: Double
sum :: {(Int -> (Int -> Int)), (Double -> (Double -> Double))}
sum(1) :: (Int -> Int)
sum(x) :: (Double -> Double)}
```

Глава 2

Шаблоны функций

Цель работы уже была коротко заявлена во введении. Она состоит в том, чтобы исследовать и реализовать механизм шаблонов функций, при котором:

1. тело шаблона подвергается максимальной проверке семантики;
2. в результате компиляции шаблона строится набор ограничений;
3. при инстанцировании шаблона повторный анализ тела не требуется — достаточно проверить ограничения.

В этой главе мы более подробно остановимся на параметрическом полиморфизме и полиморфном коде. Затем рассмотрим достоинства и проблемы предлагаемого подхода к шаблонам и его реализацию.

2.1 Обзор

Вспомним еще раз два рассмотренных подхода к шаблонам:

1. «Можно все», минимальная проверка обобщенного кода, полная проверка кода для каждой инстанции.
2. «Можно только то, что разрешено», полная проверка обобщенного кода, проверка ограничений для инстанции.

Сделаем несколько замечаний:

- когда мы находимся в теле шаблона, то не знаем, какие конкретные типы будут использованы, а значит не можем знать, какие действия над значениями этих типов разрешены;
- в теле шаблона могут быть использованы как возможности, общие для всех типов (например, применение функции `f :: S → T` к аргументу `x :: S`), так и специфические для конкретного типа (`a` **`div`** `b`);
- в теле шаблона могут находиться полностью «конкретные» выражения (то есть вообще не зависящие от типов-параметров шаблона).

При первом подходе (в языке C++) мы можем писать все, что угодно. Можем использовать глобальные функции, методы классов. Корректность их использования будет проверена лишь при инстанцировании конкретными типами. Мы можем написать обобщенный код, который будет работать для любых типов, а можем написать и такой, который не скомпилируется ни при каких конкретных типах.

При втором подходе (в .NET) обобщенный код подвергается полной проверке. Если мы хотим использовать какие-то специфические возможности типа-параметра шаблона, то должны самостоятельно явно прописать их в ограничениях. Однако, ограничения не дают полной свободы действий. К примеру, мы не сможем написать шаблон функции, в котором будет использован вызов статической перегруженной конкретной функции от параметров шаблона.

2.2 Полиморфный код, параметрический полиморфизм и реконструкция типов

Обычные функции в языках программирования имеют фиксированные типы и ведут себя одинаково в любом контексте. **Полиморф-**

ный код (в частности, **полиморфные функции**) — это код, который можно использовать с различными типами. Это свойство кода называется **полиморфизмом**.

Одним из видов полиморфизма является **специализированный полиморфизм (ad-hoc polymorphism)** [4]. К нему относится, например, *перегрузка функций*.

Ещё одна форма полиморфизма — это **параметрический полиморфизм (parametric polymorphism)** [4]. Он работает за счет использования вместо конкретных типов **типовых переменных**, которые могут быть конкретизированы «настоящими» типами. Шаблоны функций являются примером параметрического полиморфизма, а параметры шаблона выступают в роли типовых переменных.

2.2.1 Реконструкция типов

В некоторых языках (например, ML или Haskell) полное указание типов параметров функций не требуется. Типы выводятся автоматически исходя из способов использования выражений. Этот процесс называется **реконструкцией типов** (type reconstruction) (или **выводом типов**, type inference).

Замечание. В языках Haskell и ML для реконструкции типов используются алгоритмы, аналогичные **алгоритму вывода типов Хиндли-Милнера**, который в чистом виде может быть использован для типизированного лямбда-исчисления.

Подробную информацию о реконструкции типов можно найти в [4], основные понятия коротко приведем далее[1].

Примечание. Реконструкцию типов Бенджамин Пирс рассматривает на примере простого типизированного лямбда-исчисления с булевскими значениями (`Bool`) и натуральными числами (`Nat`).

[1] Бенджамин Пирс, Типы в языках программирования, Глава 22

Введем понятие **типовой переменной**, которая может быть *конкретизирована* с помощью подстановки типов. **Подстановка типов** — это конечное отображение типовых переменных на типы. Например, $[\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{U}]$ означает подстановку, сопоставляющую $\mathtt{T}$ переменной $\mathtt{X}$ и $\mathtt{U}$ переменной $\mathtt{Y}$. Запись $dom(\sigma)$ обозначает множество типовых переменных, встречающихся в левой части пар подстановки, а $range(\sigma)$ — множество типов, встречающихся в правой части. Все элементы подстановки применяются одновременно. Например, $[\mathtt{X} \mapsto \mathtt{Bool}, \mathtt{Y} \mapsto \mathtt{X} \to \mathtt{X}]$ переводит $\mathtt{X}$ в $\mathtt{Bool}$, а $\mathtt{Y}$ переводит в $\mathtt{X} \to \mathtt{X}$, а не в $\mathtt{Bool} \to \mathtt{Bool}$.

Применение подстановки к типу определяется следующим образом:

$$\begin{aligned}
\sigma(\mathtt{X}) &= \begin{cases} \mathtt{T} & \text{если } (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \\ \mathtt{X} & \text{если } \mathtt{X} \notin dom(\sigma) \end{cases} \\
\sigma(\mathtt{R}) &= \mathtt{R}, \text{где } \mathtt{R} \in \{\mathtt{Bool}, \mathtt{Nat}\} \\
\sigma(\mathtt{T}_1 \to \mathtt{T}_2) &= \sigma\mathtt{T}_1 \to \sigma\mathtt{T}_2
\end{aligned}$$

Если имеются подстановки σ и γ, то обозначим записью $\sigma \circ \gamma$ подстановку, которая получается их композицией по следующим правилам:

$$\sigma \circ \gamma = \left[\begin{array}{ll} \mathtt{X} \mapsto \sigma(\mathtt{T}) & \text{для всех } (\mathtt{X} \mapsto \mathtt{T}) \in \gamma \\ \mathtt{X} \mapsto \mathtt{T} & \text{для всех } (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \text{ при } \mathtt{X} \notin dom(\gamma) \end{array} \right]$$

Заметим, что $(\sigma \circ \gamma)\mathtt{S} = \sigma(\gamma\mathtt{S})$.

Ограничением назовем уравнение вида $\mathtt{S} = \mathtt{T}$, где $\mathtt{S}$, $\mathtt{T}$ — это базовые типы, типовые переменные или функциональные типы (содержащие как базовые типы, так и типовые переменные). Подстановка σ **унифицирует** (unifies) уравнение $\mathtt{S} = \mathtt{T}$, если результаты подстановки $\sigma\mathtt{S}$ и $\sigma\mathtt{T}$ совпадают.

Множество ограничений (constraint set) C — это набор уравнений $\{\mathtt{S}_i = \mathtt{T}_i{}^{i \in 1..n}\}$. Мы говорим, что подстановка σ **унифицирует** C (или **удовлетворяет** ему), если она унифицирует все уравнения в C.

Алгоритм реконструкции типов для простого типизированного лямбда-исчисления с несколькими базовыми типами состоит в следующем:

1. собрать множество ограничений C;
2. *унифицировать* множество ограничений;
3. если унификация прошла успешно, то на выходе получена подстановка типов, унифицирующая множество ограничений. Применение этой подстановки к выражениям (по которым были собраны ограничения) восстановит типы;
4. Если множество ограничений не унифицируется, значит где-то содержится ошибка, и типы не могут быть выведены.

2.2.2 Построение ограничений

Ограничения на типы собираются по очевидным правилам. К примеру, если некоторое выражение типа `S` используется в качестве условия конструкции `if`[2], то нужно добавить ограничение $\texttt{S} = \texttt{Bool}$ (так как в качестве условия может быть использовано только выражение логического типа).

2.2.3 Унификация

Для анализа ограничений используется **алгоритм унификации Хиндли-Милнера**, состоящий в поиске наиболее общего унификатора. **Наиболее общим унификатором** называют такую подстановку типов σ, что любая другая подстановка σ', унифицирующая множество ограничений C, может быть получена как $\sigma' = \gamma \circ \sigma$.

Указанный ниже алгоритм унификации можно найти в [4]. Выражение «пусть $\{\texttt{S} = \texttt{T}\} \cup C' = C$» во второй строке должно читаться

[2]Имеется в виду простое типизированное лямбда-исчисление с типами `Nat` и `Bool`

```
unify(C)  =  если C = ∅, то []
             иначе пусть {S = T} ∪ C' = C, и тогда
               если S = T
                 тогда unify(C')
               иначе если S = X и X ∉ FV(T)
                 тогда unify([X ↦ T]C') ∘ [X ↦ T]
               иначе если T = X и X ∉ FV(S)
                 тогда unify([X ↦ S]C') ∘ [X ↦ S]
               иначе если S = S1 →S2 и T = T1 →T2
                 тогда unify(C' ∪ {S1 = T1, S2 = T2})
               иначе
                 неудача
```

Рис. 2.1: Алгоритм унификации

как «выберем ограничение $\mathtt{S} = \mathtt{T}$ из набора ограничений C и обозначим множество оставшихся ограничений в C символом C'».

Через $FV(\mathtt{T})$ обозначим множество типовых переменных, содержащихся в $\mathtt{T}$.

В [4] доказывается, что *алгоритм unify всегда завершается*. При этом он терпит неудачу, если получает на входе невыполнимый набор ограничений, иначе возвращает наиболее общий унификатор.

В [7] также можно найти алгоритм унификации выражений. Он аналогичен приведенному выше, хотя использует другую систему типов (в частности, функции представлены в форме $(\mathtt{S}_1 \times \ldots \times \mathtt{S}_n \to \mathtt{T})$. Здесь алгоритм унификации используется для проверки корректности применения полиморфных функций. Их тип выводится по правилам, аналогичным правилам построения ограничений. В типах таких полиморфных функций возникают типовые переменные, которые связаны квантором всеобщности, например функция определения длины списка имеет

тип $\forall\alpha.list(\alpha) \to integer$. То есть это полиморфные функции, типовые переменные которых могут быть конкретизированы любым типом и не связаны никакими другими ограничениями.

2.2.4 От реконструкции типов к шаблонам функций

Следует отметить, что в результате работы алгоритма реконструкции типов выражения могут получить как конкретные типы, так и стать типизированными *типовыми переменными*.

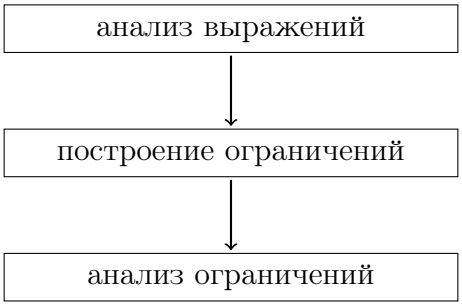

Рис. 2.2: Схема алгоритма реконструкции типов

Основываясь на идеях, лежащих в основе алгоритма реконструкции типов (см. рис. 2.2), мы можем получить желаемый механизм работы с шаблонами. Параметры шаблоны будем считать типовыми переменными. Тогда работа с шаблонами будет заключаться в следующем:

1. обойти тело шаблона, собирая ограничения на типы (при первом проходе по телу шаблона выражения могут быть типизированы и конкретными типами, и типовыми переменными-параметрами шаблона);
2. провести анализ ограничений;
3. если множество ограничений непротиворечиво, значит шаблон корректен, а на выходе получены некоторые данные, достаточные для инстанцирования (в случае простого типизированного лямбда-исчисления анализ ограничений состоял в их унификации, а результатом была подстановка типов);

4. в противном случае получена ошибка компиляции шаблона.

Наша задача состоит в том, чтобы:

1. определить *правила построения ограничений* и проверки типов внутри шаблона;
2. построить *алгоритмы анализа множества ограничений* и определить, какие данные должны быть получены на выходе;
3. определить *правила и алгоритм анализа полученных данных при инстанцировании* шаблона;

Далее будет рассмотрено конкретное решение этой задачи на примере модельного языка PollyTL.

2.3 Построение ограничений

Первый этап работы с шаблоном функции — анализ тела шаблона и построение ограничений. На этом этапе необходимо выполнить *все возможные семантические проверки* (например, использование необъявленного идентификатора или некорректное присваивание выражений конкретных типов) и, возможно, собрать ограничения на типы.

Замечание. При анализе тела шаблона может оказаться, что конкретных типов и типов-параметров шаблона недостаточно. Поэтому могут возникать новые типовые переменные.

2.3.1 Виды ограничений

Ограничение вида $\mathtt{S} = \mathtt{T}$, введенное в п. 2.2.1, назовем **базовым**. Оно возникает, например, при использовании оператора присваивания. Это «самый лучший» вид ограничений.

При вызове функции в теле шаблона будем создавать ограничение $\texttt{<name>}(\mathtt{T}_1,\ \mathtt{T}_2,\ \ldots,\ \mathtt{T}_n) :: \mathtt{R}$, которое назовем **ограничением примене-**

ния. Оно означает, что существует функция с именем `<name>`, которая может быть вызвана с n аргументами типов $\mathtt{T}_1,\ \mathtt{T}_2,\ \ldots,\ \mathtt{T}_n$, и возвращаемое значение такого вызова имеет тип `R`.

Позже нам понадобится ещё один вид ограничений — **ограничение возможных типов** (см. п. 2.4.3, с. 40). Оно имеет вид $(\mathtt{T}_1,\ \ldots,\ \mathtt{T}_n) \in \{(\mathtt{V}_{11},\ \ldots,\ \mathtt{V}_{1n}), \ldots, (\mathtt{V}_{m1},\ \ldots,\ \mathtt{V}_{mn})\}$ и означает, что истинно условие $\mathtt{T}_1 = \mathtt{V}_{i1}, \mathtt{T}_2 = \mathtt{V}_{i2}, \ldots, \mathtt{T}_n = \mathtt{V}_{in}$ для некоторого i, где $1 \leq i \leq m$.

2.3.2 Правила анализа шаблона и построения ограничений

Рассмотрим простейший случай шаблона функции — это шаблон, который не содержит вызова инстанций других шаблонов. Под *типовыми переменными* будем подразумевать типы-параметры шаблона и новые типовые переменные, которые могут появиться в процессе анализа.

Далее приведем все операторы, которые могут быть использованы внутри простейшего шаблона (то есть все возможные операторы языка, кроме вызова инстанции шаблона) и соответствующие им правила анализа и построения ограничений.

Напомним некоторые обозначения: C — множество ограничений; $FV(\mathtt{T})$ — множество типовых переменных типа `T`. Если `T` — конкретный тип, то $FV(\mathtt{T}) = \varnothing$; если это типовая переменная, то $FV(\mathtt{T}) = \{\mathtt{T}\}$; иначе это функциональный тип $\mathtt{S} \rightarrow \mathtt{U}$ и $FV(\mathtt{T}) = FV(\mathtt{S}) \cup FV(\mathtt{U})$. Через $type(\mathtt{expr})$ будем обозначать тип выражения `expr`.

Текущий контекст обозначим Γ, и будем писать $\mathtt{x} \in \Gamma (\mathtt{x} \notin \Gamma)$, если информация об идентификаторе `x` содержится (не содержится) в контексте Γ. Через Ω обозначим множество типов, известных на данный момент.

Замечание. Если в теле шаблона встречается идентификатор `<id>`, то выполняется проверка условия $\mathtt{<id>} \in \Gamma$. Если оно ложно, возникает ошибка «необъявленный идентификатор `<id>`». Если же используется

1. $\Gamma = \Gamma \cup \{$`x` $\rightarrow$ (`T`, *variable*)$\}$
2. `Пусть E` = *type*(`<expr>`). `Если E = T, то останавливаемся.`
3. `Если T это типовая переменная, то` $C = C \cup \{$`T = E`$\}$ `и останавливаемся.`
4. `Если E это типовая переменная, то` $C = C \cup \{$`E = T`$\}$ `и останавливаемся.`
5. `Если T, E — функциональные типы, то` $C = C \cup \{$`T = E`$\}$. `Иначе` *`ошибка «выражение типа E нельзя присвоить переменной типа T»`*.

Рис. 2.3: Алгоритм обработки оператора описания переменной

тип `<typename>`, то выполняется проверка условия `<typename>` $\in \Omega$.

Описание переменной с явным указанием типа. Для оператора

```
T x;
```

получаем `x :: T`, то есть $\Gamma = \Gamma \cup \{$`x` $\rightarrow$ (`T`, *variable*)$\}$, где запись `x` $\rightarrow$ (`T`, *variable*) означает, что в контекст добавлена информация о переменной `x` типа `T`.

Описание переменной с явным указанием типа и начальным значением. Если встречен оператор

```
T x = <expr>;
```

то используется алгоритм на рис. 2.3.

Замечание. Функциональные типы `T` и `E` на шаге 5 могут оказаться такими, что их равенство невозможно. Например, условие `T = E | T = Int` $\rightarrow$ `U, E = Double` $\rightarrow$ `U` невыполнимо. На данном этапе проверок не делается, но соответствующая ошибка будет обнаружена на этапе анализа ограничений.

Оператор присваивания. Пусть `x :: T`. Тогда для

```
x = <expr>;
```

нужно использовать алгоритм на рис. 2.3, выбросив шаг 1.

Описание переменной с автовыводом типа. Для оператора

```
var x = <expr>;
```

1. Пусть T = $type(\texttt{<cond}_i\texttt{>})$.
2. Если T = Bool, то останавливаемся.
3. Если T — это типовая переменная, то $C = C \cup \{\texttt{T} = \texttt{Bool}\}$. Иначе *ошибка «в качестве условия может быть использовано только выражение типа Bool»*.

Рис. 2.4: Алгоритм обработки условного выражения $\texttt{<cond}_i\texttt{>}$

$$\Gamma = \Gamma \cup \{\texttt{x} \rightarrow (type(\texttt{<expr>}),\ variable)\}$$

Условный оператор. Для проверки условных выражений $\texttt{<cond}_i\texttt{>}, i \in 0..n$ условного оператора

```
if <cond_0> then
    <statements_0>
{elif <cond_i> then
    <statements_i>}
[else
    <statements>]
fi
```

используется алгоритм на рис. 2.4.

Оператор цикла.

```
while <cond> do
    <statements>
endw
```

Для условия <cond> следует воспользоваться алгоритмом на рис. 2.4.

Вызов функции. Рассмотрим вызов функции от n аргументов:

```
<name>(x1, x2, ..., xn);
```

где $\texttt{x}_1 :: \texttt{X}_1, \texttt{x}_2 :: \texttt{X}_2, \ldots, \texttt{x}_n :: \texttt{X}_n$.

Следует сделать несколько замечаний. Во-первых, функции с именем <name> может не быть в текущем контексте. Во-вторых, если такая функция есть, то её тип либо может быть определен однозначно, либо это может быть перегруженная функция.

1. Если <name> $\notin \Gamma$, то *ошибка «неизвестное имя функции <name>»*.
2. Если <name> имеет конкретный базовый тип, то *ошибка «<name> не является именем функции или функциональной переменной»*.
3. Если $types($<name>$) = \{T\}$ (то есть тип <name> определен однозначно), то перейти к шагу 4. Иначе перейти к шагу 6.
4. (a) $k = 1$

   (b) Если $\mathtt{T} = \mathtt{S} \to \mathtt{U}$, то:

      i. Если $\mathtt{S} = \mathtt{X}_k$, то перейти к шагу 4(b)v.

      ii. Если $\mathtt{X}_k$ это типовая переменная, то $C = C \cup \{\mathtt{X}_k = \mathtt{S}\}$, переход к шагу 4(b)v.

      iii. Если S это типовая переменная, то $C = C \cup \{\mathtt{S} = \mathtt{X}_k\}$, переход к шагу 4(b)v.

      iv. Если S и $\mathtt{X}_k$ — функциональные типы, то $C = C \cup \{\mathtt{S} = \mathtt{X}_k\}$. Иначе *ошибка «в качестве аргумента с номером k ожидалось выражение типа S»*.

      v. $\mathtt{T} = \mathtt{U}$, $k = k + 1$.

      vi. Перейти к шагу 4e.

   (c) Если T это типовая переменная, то:

      i. Завести новую типовую переменную R.

      ii. $C = C \cup \{\mathtt{T} = \mathtt{X}_k \to R\}$.

      iii. $\mathtt{T} = \mathtt{R}$, $k = k + 1$.

      iv. Перейти к шагу 4e.

   (d) Если дошли до этого пункта, значит T — конкретный базовый тип. *Ошибка «вызов функции <name> не может содержать более ($k - 1$) аргументов»*.

   (e) Если $k > n$, то перейти к шагу 5, иначе перейти к шагу 4b.
5. Типизировать вызов функции <name>$(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ типом T. Остановиться.
6. Завести новую типовую переменную Z.
7. $C = C \cup \{$<name>$(\mathtt{X}_1, \ldots, \mathtt{X}_n) :: \mathtt{Z}\}$.
8. Типизировать вызов функции <name>$(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ типом Z.

Рис. 2.5: Алгоритм обработки применения функции

Тип функции определяется однозначно в двух случаях: `<name>` — это имя переменной (или формального параметра шаблона); `<name>` — это имя функции, не имеющей перегруженных версий. В первом случае в типе `<name>` могут содержаться типовые переменные. Если тип функции определен однозначно, то мы знаем типы формальных параметров и тип возвращаемого значения функции. Если же функция перегружена, то просто добавим ограничение применения, а остальные проверки отложим на более поздний этап (ведь может оказаться, например, так, что ни одна версия этой функции не может быть вызвана с таким числом аргументов).

Итак, функция вызвана от n аргументов с типами $\mathtt{X}_1, \ldots, \mathtt{X}_n$, где $n \geq 1$. **Примечание.** Вызов функции без аргументов эквивалентен вызову от одного аргумента типа `void`.

Тогда используется алгоритм на рис. 2.5.

Замечание. Для шага 4(b)iv алгоритма справедливо замечание об ограничении равенства функциональных типов (с. 30).

Таким образом мы рассмотрели основные операторы, допустимые внутри шаблона функции, кроме вложенного вызова инстанции шаблона. Об этом будет рассказано в пункте 2.5.

2.3.3 Об этапе построения ограничений

Входными данными при анализе тела шаблона являются:

1. множество известных типов Ω, включая типы-параметры шаблона;
2. контекст Γ, который содержит информацию о глобальных переменных, функциях, и формальных параметрах шаблона.

Вводим множество ограничений $C = \varnothing$. *Алгоритм* работы первого этапа заключается в применении рассмотренных выше правил к каждому оператору тела шаблона.

Результат работы алгоритма состоит в следующем:

1. получено множество ограничений $C = \{C_i \mid C_i$ — это базовое ограничение или ограничение применения$\}$;
2. все выражения в теле шаблона типизированы (конкретными типами, типами-параметрами шаблона или типовыми переменными, возникшими в процессе анализа).

Замечание. Новые типовые переменные появляются в двух случаях: (1) как тип возвращаемого значения перегруженной функции; (2) как «составная часть» функционального типа, соответствующего некоторой типовой переменной. И в том, и в другом случае новые типовые переменные сопровождаются некоторым ограничением. Если тип-параметр шаблона можно указать явно (описав переменную типа `T` | `T` — параметр шаблона), то имена типовых переменных пользователю неизвестны, а выражения типизируются данными типами неявно в процессе работы алгоритма.

2.4 Анализ ограничений

После завершения этапа построения ограничений тело шаблона может содержать ошибки. В этом случае множество ограничений C должно оказаться невыполнимым. Иначе C — достаточный набор ограничений, характеризующий шаблон функции.

2.4.1 Возможные ошибки в теле шаблона

Отметим сначала, какие ошибки *отсутствуют* к моменту начала текущего этапа.

1. Тело шаблона не содержит синтаксических ошибок.
2. Все используемые типы известны.
3. Не используются необъявленные идентификаторы.

Кроме того, если выражение (или оператор) состоит только из подвыражений конкретных типов и не содержит вызова перегруженной функции, то оно типизировано конкретным типом.

Теперь укажем ошибки, которые *может содержать* тело шаблона, и которые *должны быть выявлены* при анализе множества ограничений.

1. Выражения типа `X`, где `X` — некоторая типовая переменная, используются таким образом, что `X` не может быть типизирована ни одним конкретным типом. Например:

```
void fun t[!X]()
    X x;
    x = 5;                  // X = Int
    if x then               // X = Bool
        // do something
    fi
end

void fun p[!T](T x, T y)
    Bool b = x <= y;        // (T = Int) or (T = Double)
    var z = x(4);           // T = Int -> R, R - new type variable
end
```

   В первом случае из оператора `x = 5` следует, что `X` это на самом деле тип `Int`, а использование `x` в качестве условного выражения «обязывает» `X` быть типом `Bool`. Получается противоречие. В примере с функцией `p` из первого оператора следует, что множеством возможных конкретизаций `X` является множество {`Int`, `Double`}, а вызов `x` в качестве функции обязывает `X` быть функциональным типом. Ни `Int`, ни `Double` функциональными типам не являются. А вот следующий шаблон функции совершенно корректен:

```
S fun altApp[!F, S](F f, S x, S y)
    Int r1 = f(x);       // F = S -> Int
    var r2 = f(y);
    if r1 > r2 then
        return x;
    else
        return y;
    fi
end
```

Из оператора `Int r1 = f(x)` следует, что `F` — это функциональный тип, причем возвращаемое значение имеет тип `Int`. А вот для типа `S` никаких ограничений нет. Это может быть, например, тип `Bool`, а может быть и сколь угодно сложный функциональный тип.

2. Некорректный вызов перегруженной функции. Это может быть вызов с лишним числом параметров или вызов с некорректными типами (ведь на предыдущем шаге для перегруженных функций никаких проверок не проводилось).
3. Возвращаемый тип перегруженной функции не может быть определен ни при каких конкретизациях шаблона.

   Рассмотрим два примера. Пусть определены 4 версии перегруженной функции `sum` с типами: `Int -> (Int -> Int)` (т.е. функция может быть вызвана не более, чем с двумя параметрами), `Int -> (Int -> (Int -> Int))` (функция может быть вызвана не более, чем с тремя параметрами) и двумя аналогичными для типа Double.

```
fun good[!T](T x, T y, T z)
    var r = sum(x, y, z);
end

fun bad[!T](T x, T y)
    var r = sum(x, y);
end

fun notbad1[!T](T x, T y, T z)
    T r1 = sum(x, y);
    T -> T r2 = sum(x, y);
end
fun notbad2[!T, R](T x, T y, T z)
    R r = sum(x, y);
end
```

В функции `good1` есть две подходящих версии для вызова `sum`: с тремя аргументами типа Int, и тремя аргументами типа Double. В зависимости от типа `T` при инстанцировании шаблона будет выбрана нужная.

Посмотрим на функцию `bad`: подходят все 4 версии `sum`. А что будет при инстанцировании шаблона? Типовая переменная `T` может быть конкретизирована либо типом `Int`, либо типом `Double`. При этом в обоих случаях для вызова `sum` останутся два варианта: вызывать `sum` с двумя параметрами и базовым возвращаемым типом, или вызывать `sum` с функциональным возвращаемым типом (так как имеет место частичное применение функции, и `f(x) :: T -> T`, если `f :: T -> (T -> T)`, `x :: T`). Значит независимо от допустимой конкретизации типовой переменной `T` получаем неоднозначность: тип выражения `sum(x, y)` не может быть выведен. Исправить эту ситуацию можно — для этого нужно указать возвращаемый тип вызова `sum`:

(a) если на момент написания шаблона известно, какого характе-

ра должен быть возвращаемый тип при вызове `sum`, то можно воспользоваться моделью функции `notbad1`;

(b) если такой информации нет, то решение можно отложить на этап инстанцирования шаблона, указав в качестве возвращаемого типа параметр шаблона `R` (как в функции `notbad2`).

Далее перейдем к процессу анализа множества ограничений, который будет состоять из нескольких этапов.

2.4.2 Первый этап: унификация базовых ограничений

Напомним, что **базовым** мы называем ограничение вида `S` = `T`, где `S, T` — это базовые типы, типовые переменные или функциональные типы (содержащие как базовые типы, так и типовые переменные). Через $FV(\texttt{T})$ обозначается множество типовых переменных, содержащихся в типе `T`.

Множество ограничений C состоит из базовых ограничений и ограничений применения. Разобьем его на два подмножества: $C = C^b \cup C^{app}$, где C^b — множество базовых ограничений, C^{app} — множество ограничений применения, и, очевидно, $C^b \cap C^{app} = \varnothing$.

Поскольку базовые ограничения соответствуют ограничениям, указанным в п. 2.2.1, мы можем воспользоваться алгоритмом унификации на рис. 2.1 (с. 26, п. 2.2.3) для множества C^b. Он приведен ещё раз на рис. 2.6.

Как уже было отмечено, алгоритм унификации всегда завершается. На выходе он дает *наиболее общий унификатор множества* C^b, если множество C^b выполнимо, и завершается неудачей в противном случае.

Множество C^b было получено на этапе построения ограничений из:

- операторов вызова функции с однозначно определенным типом;
- описания переменных с указанием начальных значений;
- операторов присваивания;

```
unify(C^b) = если C^b = ∅, то []
             иначе пусть C^b = {S = T} ∪ C'^b, и тогда
               если S = T
                 тогда unify(C'^b)
               иначе если S = X и X ∉ FV(T)
                 тогда unify([X ↦ T]C'^b) ∘ [X ↦ T]
               иначе если T = X и X ∉ FV(S)
                 тогда unify([X ↦ S]C'^b) ∘ [X ↦ S]
               иначе если S = S1 →S2 и T = T1 →T2
                 тогда unify(C'^b ∪ {S1 = T1, S2 = T2})
               иначе
                 ошибка «невозможно удовлетворить ограничение (S = T)»
```

Рис. 2.6: Алгоритм унификации

- использования выражений в качестве условий операторов `if` и `while`.

Обозначим результат унификации C^b как

$$\sigma = unify(C^b)$$

Если унификация прошла успешно, то σ содержит подстановку типов, унифицирующую исходное множество C^b. Значит на данный момент в указанных выше операторах ошибок не обнаружено. В свою очередь это означает, что отловлена часть ошибок категории 1 (с. 35).

На рис. 2.7 приведен алгоритм работы данного этапа.

Вход: C

Алгоритм:

$C = C^b \cup C^{app}$

$\sigma = unify(C^b)$

Выход: C^{app}, σ

Рис. 2.7: Алгоритм работы на этапе унификации базовых ограничений

2.4.3 Второй этап: преобразование ограничений применения

Вводные замечания

Ограничением применения мы назвали ограничение следующего вида: `<name>`($\mathtt{X}_1$, $\mathtt{X}_2$, …, $\mathtt{X}_n$) :: R. Оно означает, что существует функция с именем `<name>`, которая может быть вызвана с n аргументами типов $\mathtt{X}_1$, $\mathtt{X}_2$, …, $\mathtt{X}_n$, и возвращаемое значение такого вызова имеет тип R.

Ограничения данного вида строились по оператору вызова перегруженной функции, при этом мы даже не были уверены, что существует хоть одна версия функции `<name>`, которая может быть вызвана с n аргументами.

Таким образом на данном этапе необходимо *проверить корректность ограничений применения* $c \in C^{app}$. Возможны следующие ситуации:

1. нет ни одной перегруженной версии функции, которая может быть вызвана с указанными типами параметров и типом возвращаемого значения[3] ⇒ ошибка компиляции шаблона;
2. среди экземпляров перегруженной функции есть только один подходящий экземпляр ⇒ станут известны конкретизации типовых переменных, содержащихся в ограничении;
3. есть несколько подходящих версий перегруженной функции ⇒ для типовых переменных текущего ограничения применения известны множества возможных конкретизаций.

Напомним, что ограничение применения всегда содержит хоть одну типовую переменную — тип возвращаемого значения функции. Это следует из правил построения ограничений (см. шаги 6–7 алгоритма на рис. 2.5, с. 32)

[3] см. замечание*, с. 41

Замечание*. Ясно, что в качестве типов параметров и возвращаемого значения функции могут быть указаны типовые переменные, в то время как перегруженные функции имеют конкретные типы. Это нужно учесть при анализе ограничений применения.

Ограничением возможных типов мы назвали ограничение вида $(\mathtt{T}_1,\ \ldots,\ \mathtt{T}_n) \in \{(\mathtt{V}_{11},\ \ldots,\ \mathtt{V}_{1n}), \ldots, (\mathtt{V}_{m1},\ \ldots,\ \mathtt{V}_{mn})\}$, которое означает, что истинно условие $\mathtt{T}_1 = \mathtt{V}_{i1}, \mathtt{T}_2 = \mathtt{V}_{i2}, \ldots, \mathtt{T}_n = \mathtt{V}_{in}$ для некоторого i, где $1 \leq i \leq m$.

Преобразование ограничений применения

На предыдущем шаге было получено множество ограничений применения C^{app} и подстановка типов σ. Поскольку ограничения $c \in C^{app}$ могут содержать типовые переменные $\mathtt{X} \in dom(\sigma)$, необходимо применить к ним подстановку σ. Расширим введенное ранее определение *применения подстановки*:

$$
\begin{array}{lcl}
\sigma(\mathtt{X}) & = & \begin{cases} \mathtt{T} & \text{если } (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \\ \mathtt{X} & \text{если } \mathtt{X} \notin dom(\sigma) \end{cases} \\
\sigma(\mathtt{V}) & = & \mathtt{V}, \text{где } \mathtt{V} \in \{\mathtt{Bool}, \mathtt{Int}, \mathtt{Double}\} \\
\sigma(\mathtt{T}_1 \rightarrow \mathtt{T}_2) & = & \sigma\mathtt{T}_1 \rightarrow \sigma\mathtt{T}_2 \\
\sigma(\texttt{<name>}(\mathtt{X}_1,\ \mathtt{X}_2,\ \ldots,\ \mathtt{X}_n) :: \mathtt{R}) & = & \texttt{<name>}(\sigma\mathtt{X}_1,\ \sigma\mathtt{X}_2,\ \ldots,\ \sigma\mathtt{X}_n) :: \sigma\mathtt{R}
\end{array}
$$

И применим подстановку σ ко множеству ограничений применения:

$$C^{app} = \sigma C^{app}$$

Через $types(\texttt{<name>})$ обозначено множество типов перегруженных версий функции `<name>`.

Пусть `rtypes` и `ctypes` это наборы конкретных типов, $|\mathtt{ctypes}| = |\mathtt{rtypes}| = N$, элементы данных наборов пронумерованы натуральными числами $i \in 1..M, M \geq N$. Запись $\mathtt{T}_i \in \mathtt{ctypes}$ означает, что набор

$\texttt{ctypes}$ содержит элемент с номером i. Наборы таковы, что $\texttt{T}_i \in \texttt{ctypes} \Leftrightarrow \texttt{S}_i \in \texttt{rtypes}$.

Функция $apply\_type(\texttt{V}, \texttt{rtypes}, \texttt{ctypes})$ (где $\texttt{V}$ — конкретный тип) возвращает пару наборов $(\texttt{rtypes}', \texttt{ctypes}')$, где

$$\texttt{rtypes}' = \{\texttt{S}_i \mid \texttt{T}_i \in \texttt{rtypes}, \texttt{T}_i = \texttt{V} \rightarrow \texttt{S}_i\}$$
$$\texttt{ctypes}' = \{\texttt{T}_i \mid \texttt{T}_i \in \texttt{ctypes}, \texttt{S}_i \in \texttt{rtypes}'\}$$

причем $|\texttt{ctypes}'| = |\texttt{rtypes}'| = N', 0 \leq N' \leq N$ и $\texttt{T}_i \in \texttt{ctypes}' \Leftrightarrow \texttt{S}_i \in \texttt{rtypes}'$.

Функция $miss\_type(\texttt{rtypes}, \texttt{ctypes})$ возвращает пару наборов $(\texttt{rtypes}', \texttt{ctypes}')$, где

$$\texttt{rtypes}' = \{\texttt{S}_i \mid \texttt{T}_i \in \texttt{rtypes}, \texttt{T}_i = \texttt{X} \rightarrow \texttt{S}_i \text{ для некоторого } \texttt{X}\}$$
$$\texttt{ctypes}' = \{\texttt{T}_i \mid \texttt{T}_i \in \texttt{ctypes}, \texttt{S}_i \in \texttt{rtypes}'\}$$

и для наборов $\texttt{ctypes}', \texttt{rtypes}'$ имеют место те же ограничения, что и в предыдущем случае.

То есть функции обрабатывают набор типов $\texttt{rtypes}$, оставляя типы, удовлетворяющие некоторому условию, и одновременно выбрасывают из $\texttt{ctypes}$ типы, номера которых не попали в $\texttt{rtypes}'$.

Тогда на рис. 2.8 приведен алгоритм обработки ограничения применения $process\_app\_cons(c^{app}), c^{app} \in C^{app}$.

1. $c^{app} =$ <name>$(\mathtt{X}_1,\ \mathtt{X}_2,\ \ldots,\ \mathtt{X}_n) :: \mathtt{R}$
2. $N = types($<name>$)$
3. $\mathtt{ctypes} = numerate(types($<name>$), 1, N)$, $\mathtt{rtypes} = \mathtt{ctypes}$
4. $k = 1$
5. Если $\mathtt{X}_k$ — это конкретный тип (то есть $FV(\mathtt{X}_k) = \varnothing$), то переход к шагу 6. Иначе переход к шагу 8.
6. $(\mathtt{rtypes}, \mathtt{ctypes}) = apply\_type(\mathtt{X}_k, \mathtt{rtypes}, \mathtt{ctypes})$
7. Если $|\mathtt{rtypes}| = 0$, то *ошибка «ни одна перегруженная версия функции <name> не может быть вызвана с параметром номер k типа $\mathtt{X}_k$».* Иначе переход к шагу 10.
8. Известно, что $FV(\mathtt{X}_k) \neq \varnothing$, то есть $\mathtt{X}_k$ не является конкретным типом (см. замечание*).
   $(\mathtt{rtypes}, \mathtt{ctypes}) = miss\_type(\mathtt{rtypes}, \mathtt{ctypes})$
9. Если $|\mathtt{rtypes}| = 0$, то *ошибка «ни одна перегруженная версия функции <name> не может быть вызвана более чем с $(k-1)$ параметрами».*
10. $k = k + 1$
11. Если $k \leq n$, то переход к шагу 5.
12. Если $|\mathtt{ctypes}| = 1$, то 12a, иначе ($|\mathtt{ctypes}| \geq 2$) 12b.

    (a) $\mathtt{T} \in \mathtt{ctypes}$ является подходящим экземпляром перегруженной функции <name> и имеет вид $\mathtt{V}_1 \to (\mathtt{V}_2 \to (\ldots \to (\mathtt{V}_n \to \mathtt{U})\ldots))$, $\mathtt{V}_i$, $\mathtt{U}$ — конкретные типы. Делаем вывод о равенстве соответствующих типов и возвращаем множество базовых ограничений
    $$C_0^b = \{\mathtt{X}_i = \mathtt{V}_i\}_{i=1}^n \cup \{\mathtt{R} = \mathtt{U}\}$$

    (b) Рассмотрим полученный набор $\mathtt{ctypes} = \{W_{i_j}\}, i_j \in 1..N, j \in 1..m_0, 2 \leq m_0 \leq N$. Множество возможных индексов i_j обозначим $I^{\mathtt{ctypes}}$. $W_{i_j} = \mathtt{V}_{i_j,1} \to (\mathtt{V}_{i_j,2} \to (\ldots \to (\mathtt{V}_{i_j,n} \to \mathtt{U}_{i_j})\ldots))$. Возвращаем ограничение возможных типов
    $$c_0^{tp} = correct\_ctp((\mathtt{X}_1,\ \ldots,\ \mathtt{X}_n, \mathtt{R}) \in \{(\mathtt{V}_{i_j,1},\ \ldots,\ \mathtt{V}_{i_j,n},\ \mathtt{U}_{i_j})\}_{i_j \in I^{\mathtt{ctypes}}})$$

Рис. 2.8: Алгоритм обработки ограничения применения $process\_app\_cons(c^{app})$

Замечание*. Алгоритм поиска подходящих версий перегруженной функции таков, что: если тип некоторого параметра $\mathtt{X}_k$ не является

полностью конкретным, то на шаге 8 алгоритма считается, что он может быть сопоставлен с соответствующим конкретным типом $\mathtt{V}_k$. То есть, к примеру, для $\mathtt{X}_k = \mathtt{T} \to \mathtt{Int}, \mathtt{V}_k = \mathtt{Double}$ соответствующая типу $\mathtt{V}_k$ перегруженная версия не будет отброшена.
Алгоритм работы таков, что базовые ограничения, полученные на шаге 12a, впоследствии будут унифицированы с помощью алгоритма на рис. 2.6, и, следовательно, указанное допущение будет проверено.
Что касается шага 12b — для построенного ограничения возможных типов проводится дополнительная проверка (функция $correct\_ctp$). На выходе получается:

- либо ограничение возможных типов $(\mathtt{T}_1,\ \ldots,\ \mathtt{T}_n) \in \{(\mathtt{V}_{i1},\ \ldots,\ \mathtt{V}_{in})\}_{i=1}^{m}$ | $\mathtt{T}_j$ сопоставимо с $\mathtt{V}_{ij}$ $\forall i$ и все $\mathtt{T}_j$ различны.
- либо *ошибка*, если в исходном ограничении выполняется условие $\nexists i\ \forall k, l\ (1 \leq k, l \leq n)\ \mathtt{T}_k = \mathtt{T}_l \Rightarrow \mathtt{V}_{ik} = \mathtt{V}_{il}$ (то есть возможные типы во всех наборах противоречивы) или $\forall i\ \exists j\ \mid \mathtt{T}_j$ не сопоставимо с $\mathtt{V}_{ij}$ (то есть множество возможных типов набора $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ пусто).

Расширим применение подстановки на ограничение возможных типов:

$$\sigma((\mathtt{T}_1,\ \ldots,\ \mathtt{T}_n) \in \{(\mathtt{V}_{i1},\ \ldots,\ \mathtt{V}_{in})\}_{i=1}^{m}) = correct\_ctp(c^{app}),$$
$$\text{где } c^{app} = (\sigma\mathtt{T}_1,\ \ldots,\ \sigma\mathtt{T}_n) \in \{(\mathtt{V}_{i1},\ \ldots,\ \mathtt{V}_{in})\}_{i=1}^{m}$$

На рис. 2.9 приведен полный алгоритм преобразования множества ограничений применения C^{app}.

$process\_apps(C^{app}, C^{tp}, \sigma)$ = если $C^{app} = \varnothing$, то (C^{tp}, σ)
иначе пусть $C^{app} = c^{app} \cup C'^{app}$, и тогда
$C_0 = process\_app\_cons(c^{app})$ (возможна *ошибка*)
если $C_0 = C_0^b$ (множество базовых ограничений), тогда
$\sigma' = unify(C_0^b)$ (возможна *ошибка*)
$process\_apps(\sigma' C'^{app}, \sigma' C^{tp}, \sigma' \circ \sigma)$
иначе $C_0 = \{c^{tp}\}$, и тогда
$process\_apps(C'^{app}, C^{tp} \cup c^{tp}, \sigma)$

Рис. 2.9: Алгоритм обработки множества ограничений применения

Заметим, что исходное множество ограничений применения C^{app} конечно и на каждом шаге уменьшается на один элемент $\Rightarrow$ алгоритм $process\_apps$ завершается, а на выходе получается конечное множество ограничений C^{tp}.

Тогда алгоритм работы текущего этапа приведен на рис. 2.10.

Вход: C^{app}, σ
Алгоритм:
$C^{tp} = \varnothing$
$(C^{tp}, \sigma) = process\_apps(C^{app}, C^{tp}, \sigma)$
Выход: C^{tp}, σ

Рис. 2.10: Алгоритм работы на этапе преобразования ограничений применения

Заметим, что в процессе работы данного этапа обрабатываются ошибки категорий 1 и 2 (с. 35).

2.4.4 Третий этап: анализ ограничений возможных типов

Предварительная обработка

Обозначим функцию $process\_ctp$, которая принимает на вход ограничение возможных типов c^{tp}. Если c^{tp} таково, что можно однозначно

вывести значения некоторых типов (к примеру, из ограничения $(\texttt{T},\ \texttt{S}) \in \{(\texttt{Int},\ \texttt{Double}), (\texttt{Int},\ \texttt{Int})\}$ следует, что $\texttt{T} = \texttt{Int}$), то функция возвращает множество соответствующих базовых ограничений C_0^b и, возможно, упрощенное ограничение возможных типов c'^{tp} (в котором отсутствуют выражения для типов из C_0^b), иначе возвращает $\{c^{tp}\}$.

Множество ограничений C^{tp} должно выполняться одновременно. Обозначим через $dom(c)$ множество типовых переменных, встречающихся в левой части ограничения $c \in C^{tp}$, а через $range(c, \texttt{X})$ — множество возможных типов типовой переменной $\texttt{X} \in dom(c)$. И введем множество $dom(C^{tp}) = \bigcup\limits_{c \in C^{tp}} dom(c)$. Тогда необходимым условием выполнимости множества ограничений C^{tp} является:

$$\begin{aligned} &\forall \texttt{X} \in dom(C^{tp})\ possible\_types(\texttt{X}) \neq \varnothing, \text{ где} \\ &possible\_types(\texttt{X}) = \bigcap_{c \in C^{tp} | \texttt{X} \in dom(c)} range(c, \texttt{X}) \end{aligned} \tag{2.1}$$

То есть, если $C^{tp} = \{c_1, c_2\}, c1 = (\texttt{T},\ \texttt{S}) \in \{(\texttt{Int},\ \texttt{Double}), (\texttt{Int},\ \texttt{Double} \to \texttt{Double})\}, c_2 = (\texttt{Y},\ \texttt{S}) \in \{(\texttt{Bool},\ \texttt{Bool}), (\texttt{Int},\ \texttt{Int})\}$, то $r_1 = range(c_1, \texttt{S}) = \{\texttt{Double}, \texttt{Double} \to \texttt{Double}\}, r_2 = range(c_2, \texttt{S}) = \{\texttt{Bool}, \texttt{Int}\}$ и $possible\_types(\texttt{S}) = r1 \cap r2 = \varnothing$. Значит *ошибка «множество возможных типов типовой переменной $\texttt{S}$ оказалось пустым»*.

Замечание** Ясно, что условие 2.1 не является достаточным. Ведь если $|dom(c)| > 1$, то ограничение c показывает зависимость между несколькими типами. И, значит, левые части ограничений могут «пересекаться» по нескольким типам одновременно. В этом случае может оказаться, что хотя множество возможных типов отдельных типовых переменных и не пусто, но одновременное выполнение требований невозможно.

Например:

```
fun di1(Double a, Int b, Int c)
    // do something
end
fun di1(Int a, Double b, Double c)
    // do something
end

fun di2(Int a, Int b, Int c)
    // do something
end
fun di2(Double a, Double b, Double c)
    // do something
end

fun f[!T, S, R, U]()
  T t;
  S s;
  R r;
  U u;
  di1(s, t, r);
  di2(u, s, r);
end
```

Множество ограничений шаблона функции `f1` имеет вид $\{$`(U, S, T)` $\in \{$`(Int, Int, Int)`, `(Double, Double, Double)`$\}$, `(S, T, R)` $\in \{$`(Double, Int, Int)`, `(Int, Double, Double)`$\}\}$. То есть из первого ограничения следует, что типовые переменные `(S, T)` могут иметь типы `(Int, Int)` либо `(Double, Double)`, а из второго ограничения — `(Double, Int)` либо `(Int, Double)`.

Задача поиска таких противоречий *на этом этапе* представляет отдельный интерес, но в данной работе не исследуется. Тем не менее, противоречие будет обнаружено на заключительном шаге анализа ограничений (см. алгоритм на рис. 2.13).

На рис. 2.11 приведен алгоритм предварительной обработки множе-

ства ограничений возможных типов C^{tp}. Под σ понимается подстановка, полученная на предыдущем этапе. $C_r^{tp} = \varnothing$.

---

1. Если $C^{tp} = \varnothing$, то остановиться.
2. Пусть $C^{tp} = c \cup C'^{tp}$
3. $C_0 = process\_ctp(c)$
4. Если $C_0 = \{c\}$, то $C_r^{tp} = C_r^{tp} \cup \{c\}$ и переход к шагу 1.
5. Иначе $C_0 \not\ni c, C_0 = C_0^b \cup C_0^{tp}$, $C_0^b = \{c^b \mid c^b$ — базовое ограничение$\}$
6. Если $C_0^{tp} \neq \varnothing$ (то есть $C_0^{tp} = \{c'\}$), то $C_r^{tp} = C_r^{tp} \cup \{c'\}$
7. $\sigma' = unify(C_0^b)$ (возможна *ошибка*)
8. $\sigma = \sigma' \circ \sigma$
9. $C^{tp} = C'^{tp} \cup C_r^{tp}$, $C_r^{tp} = \varnothing$
10. $C^{tp} = \sigma' C^{tp}$ (возможна *ошибка*)
11. Переход к шагу 1

Рис. 2.11: Алгоритм обработки ограничений $process\_types\_cons(C^{tp})$

Данный алгоритм позволяет обнаружить ошибки категории 1 (см. с. 35).

Словами его можно описать так: берем ограничение возможных типов c и пытаемся «вытащить» из него базовые ограничения. Если не удалось, то помещаем ограничение во множество рассмотренных C_r^{tp}. Иначе находим подстановку типов, вносим «остаток» ограничения c' (его может не быть, но если c' есть, то оно меньше исходного c) во множество рассмотренных ограничений C_r^{tp}. Так как получена подстановка типов, объединим C_r^{tp} и C^{tp} ($C^{tp} = C'^{tp} \cup C_r^{tp}$, $C_r^{tp} = \varnothing$) и применим подстановку к полученному множеству (в результате применения подстановки может обнаружиться противоречие, либо часть ограничений может упроститься). Снова берем некоторое ограничение из C^{tp} и обрабатываем. И так далее до тех пор, пока не опустошим множество C^{tp}.

Рассмотрим объединение множеств $C^{tp} \cup C_r^{tp}$. На каждой итерации алгоритма оно либо упрощается (шаг 5, будет найдена подстановка), ли-

бо остается неизменным (шаг 4). Упрощаться бесконечно ограничения не могут (так как левая часть ограничений возможных типов конечна, а упрощение происходит «вытаскиванием» типовых переменных из левой части), значит на какой-то итерации множество станет постоянно неизменным. С этого момента на каждой следующей итерации мы будем переносить элемент из множества C^{tp} в C^{tp}_r. Множество C^{tp} конечно, и, следовательно, алгоритм завершается. Причем на выходе мы получаем новую подстановку типов σ и набор ограничений возможных типов C^{tp}_r.

Заключительный анализ

Возможны две ситуации:

1. шаблон корректен *при любых* конкретизациях параметров;
2. шаблон корректен *только при некоторых* конкретизациях параметров;

Примером первого случая является шаблон функции `app`, примером второго — `sum`.

```
R fun app[!S, R](S -> R f, S x)
    return f(x);
end

T fun sum[!T](T x, T y)
    return x + y;
end
```

Два примера допустимых шаблонов функций

Нас интересует вопрос:

Существуют ли конкретизации параметров шаблона, при которых шаблон является корректным? (2.2)

Чтобы ответить на этот вопрос, надо разрешить две проблемы:

1. код шаблона может содержать ошибки категории 3 (см. с. 36), то есть возможно наличие такого вызова перегруженной функции, что невозможно определить, какой именно экземпляр должен быть вызван;
2. множество ограничений возможных типов может оказаться невыполнимым (см. замечание**, с. 46).

Заметим, что обе проблемы стоят только в том случае, если множество ограничений возможных типов $C_r^{tp} \neq \varnothing$. Действительно, второй проблемы нет по определению, так как пустое множество выполнимо. Что касается неоднозначного вызова перегруженной функции: если тело шаблона содержит вызов перегруженной функции, то во множество ограничений добавляется ограничение применения. Далее, при обработке этого ограничения либо обнаруживается ошибка, либо определяется нужный экземпляр, либо (если осталось несколько подходящих версий перегруженной функции) оно преобразуется в ограничение возможных типов и попадает во множество C_r^{tp}. Таким образом, если $C_r^{tp} = \varnothing$, то неоднозначных вызовов перегруженных функций нет.

Подготовим шаблон функции к заключительному анализу, применив алгоритм на рис. 2.12. Через Δ обозначено множество типов-параметров шаблона.

Входные данные:

- подстановка типов σ
- множество ограничений возможных типов C_r^{tp}, возможно пустое.

Алгоритм:

1. применить подстановку σ ко всем типам-параметрам шаблона: $\Delta' = \sigma\Delta = \{\sigma\mathtt{T} \mid \mathtt{T} \in \Delta\}$;
2. обновить информацию о типах в контексте.

Замечание. В результате применения подстановки некоторые параметры шаблона могут «превратиться» в более сложные типовые выражения или вовсе оказаться конкретизированными. Например, в следующем примере `F` $\mapsto$ `Bool` $\to$ `Y`, а `X` $\mapsto$ `Bool`.

```
fun temp[!F, X](F f, X x)
    var y = f(x);
    if x then
        // do dmth with y;
    fi
    // do smth
end
```

Рис. 2.12: Подготовка шаблона к применению заключительного алгоритма анализа

Предположим, что мы инстанцируем шаблон. Если множество ограничений $C_r^{tp} = \varnothing$, то достаточно сопоставить указанные конкретные типы типовым выражениям из Δ'. Это легко сделать, применив алгоритм унификации на рис. 2.6 (см. с. 39) ко множеству базовых ограничений $C_{inst}^b = \{\mathtt{T} = \mathtt{V} \mid \mathtt{T} \in \Delta', \mathtt{V}$ — соответствующий `T` конкретный тип$\}$, то есть $\sigma = unify(C_{inst}^b) \circ \sigma$. Если унификация прошла успешно, значит инстанция шаблона корректна.

Пусть теперь $C_r^{tp} \neq \varnothing$. Нужно дополнительно проверить, что параметры инстанции удовлетворяют множеству ограничений C_r^{tp}. Для этого применим полученную подстановку σ ко множеству ограничений C_r^{tp} и переобозначим $C_r^{tp} = \sigma C_r^{tp}$. Обработаем его с помощью алгоритма $process\_types\_cons(C_r^{tp})$ (рис. 2.11, с. 48). Если произошла ошибка, значит для данной инстанции произошла *ошибка «множество ограничений противоречиво»* (проблема 2). Если после обработки $C_r^{tp} \neq \varnothing$, значит *ошибка «не удалось вывести все типы»* (проблема 1).

Исходя из этих соображений получаем заключительный алгоритм анализа набора ограничений возможных типов C_r^{tp} (рис. 2.13), который отвечает на вопрос 2.2. Через $\Phi = dom(\Delta')$ обозначим множество типовых переменных, которые конкретизируются при инстанцировании шаблона. Под символом $\sum$ понимается дизъюнкция.

$analyze\_tpcs(C_r^{tp}, \sigma, \Phi)$ = `если` $C_r^{tp} = \varnothing$, `то` $true$
`иначе`
`если` $\Phi = \varnothing$, `то` $false$ (*невозможно вывести все типы*)
`иначе`
$\Phi = \{\mathtt{X}\} \cup \Phi'$
`вернуть` $\sum_{\mathtt{V} \in possible\_types(\mathtt{X})} analyze\_tpcs(C_r'^{tp}, \sigma', \Phi')$, `где`
$(C_r'^{tp}, \sigma') = process\_types\_cons([\mathtt{X} \mapsto \mathtt{V}]C_r^{tp}, [\mathtt{X} \mapsto \mathtt{V}] \circ \sigma)$

Рис. 2.13: Заключительный алгоритм анализа набора ограничений возможных типов

Замечание. Если на шаге получения $C_r'^{tp}$ происходит ошибка *«множество ограничений противоречиво»*, то соответствующий элемент дизъюнкции равен $false$ (при программировании это означает перехват исключения). Чтобы по результатам $analyze\_tpcs(C_r^{tp}, \sigma, \Phi)$ выдавать правильное сообщение об ошибке (неоднозначность при выводе типов/невыполнимое множество ограничений), нужно в процессе работы алгоритма следить за причиной ответа $false$.

Таким образом алгоритм *analyze_tpcs* проверяет, существуют ли корректные инстанции шаблона. Подстановка σ на момент ответа *true* как раз и характеризует эти инстанции (к примеру, для функции `app` $\sigma = [\mathtt{F} \mapsto \mathtt{X} \to \mathtt{Y}]$).

```
fun app[!F, X](F f, X x)
    return f(x);
end
```

Чем больше множество C_r^{tp}, тем более конкретными будут допустимые инстанции шаблона.

Полный алгоритм работы данного этапа анализа ограничений приведен на рис. 2.14.

Вход: C^{tp}, σ

Алгоритм:

1. $process\_types\_cons(C^{tp}) \Rightarrow C_r^{tp}, \sigma'$
2. Подготовить шаблон к применению заключительного алгоритма анализа $\Rightarrow \Phi$
3. $analyze\_tpcs(C_r^{tp}, \sigma', \Phi) \Rightarrow$ успех/неуспех

Выход:

- либо *ошибка*;
- либо множество ограничений C_r^{tp} и подстановка σ', характеризующие шаблон.

Рис. 2.14: Алгоритм работы этапа анализа ограничений возможных типов

Если алгоритм сработал успешно, то подстановку σ' нужно применить к типам аргументов шаблона функции, а так же типам выражений в теле шаблона. Полученный набор ограничений C_r^{tp} следует сохранить.

Замечание №1. Пустым может оказаться не только набор ограничений C_r^{tp}, но и подстановка σ'. Простейший пример соответствующего шаблона функции — тождественная функция `id`.

```
T fun id[!T](T x)
    return x;
end
```

Замечание №2. Вспомним директивы компилятора `#print_type` и `#print_type1` (см. п. 1.3). Вне шаблонов они работают одинаково и выводят тип выражений. А вот внутри шаблона директива `#print_type1` выводит исходный тип выражений, а `#print_type` — тип выражений после применения подстановки σ' к телу и аргументам шаблона функции.

2.5 Инстанцирование шаблона

Результатом компиляции шаблона является множество ограничений возможных типов C_r^{tp} (возможно, пустое) и подстановка типов σ (возможно, пустая). На самом деле подстановка типов уже не представляет большого интереса, так как на заключительном этапе компиляции шаблона применяется ко всем его типам, включая типы-параметры шаблона.

Таким образом в качестве результата достаточно сохранить множество C_r^{tp}. Через Δ обозначим множество типовых выражений, которые соответствуют исходным типам-параметрам шаблона Δ^0 ($\Delta = \{\mathtt{E} \mid \mathtt{E} = \sigma\mathtt{T}, \mathtt{T} \in \Delta^0\}$).

Замечание. Мы говорим «типовых выражений», так как подстановка применяется и к параметрам шаблона. Например, для функции `app` типовая переменная-параметр шаблона $\mathtt{F} \in \Delta^0$ заменится типовым выражением $\mathtt{X} \to \mathtt{Y} \in \Delta$, где `Y` — новая типовая переменная.

```
fun app[!F, X](F f, X x)
```

```
    return f(x);
end
```

То есть после компиляции шаблон функции `app`, фактически, выглядит следующим образом:

```
Y fun app[!X -> Y, X](X -> Y f, X x)
    return f(x);
end
```

Замечание. В предыдущем разделе мы вводили множество $\Phi = dom(\Delta)$. В данном примере $\Delta^0 = \{\mathtt{F}, \mathtt{X}\}, \Delta = \{\mathtt{X} \to \mathtt{Y}, \mathtt{X}\}, \Phi = \{\mathtt{X}, \mathtt{Y}\}$.

Пусть есть некоторый откомпилированный шаблон `temp` с параметрами шаблона $\Delta^0 = \{\mathtt{T}_1, \ldots, \mathtt{T}_k\}, k \geq 1$, выражениями параметров шаблона $\Delta = \{\mathtt{E}_1, \ldots, \mathtt{E}_k\}$ и типами аргументов $\mathtt{P}_1, \ldots, \mathtt{P}_n, n \geq 1$ (помним, что отсутствие аргументов эквивалентно одному аргументу типа `void`). Ясно, что $\bigcup\limits_{i \in 1..n} FV(\mathtt{P}_i) \subseteq dom(\Delta)$, так как в типах аргументов могут быть использованы не все заявленные типы-параметры шаблона.

И пусть происходит вызов шаблона в виде

$$\mathtt{temp[!T}_{i_1} = \mathtt{U}_1, \ldots, \mathtt{T}_{i_l} = \mathtt{U}_l\mathtt{]}(\mathtt{V}_1, \ldots, \mathtt{V}_m), 0 \leq l \leq k, 1 \leq m \leq n,$$

где все $\mathtt{V}_i$ — конкретные типы (рассматриваем инстанцирование шаблона в конкретном контексте). Тогда алгоритм проверки инстанции шаблона приведен на рис. 2.15.

обозначим $C^b = \{\mathtt{E}_{i_j} = \mathtt{U}_j\}_{j=1}^{l} \cup \{\mathtt{P}_j = \mathtt{V}_j\}_{j=1}^{m}$
если $\gamma = unify(C^b)$ завершилось ошибкой, то
  ошибка «не удалось инстанцировать, множество ограничений противоречиво»
иначе
  $C_r'^{tp} = \gamma C_r^{tp}$
  если применение подстановки завершилось ошибкой, то
    ошибка «не удалось инстанцировать, множество ограничений противоречиво»
  иначе
    если $\Phi' \neq \varnothing$, где $\Phi' = dom(\gamma\Delta)$, то
      ошибка «не удалось вывести все параметры шаблона»
    иначе
      если $C_r'^{tp} \neq \varnothing$, то
        ошибка «не удалось вывести все типы »
      иначе инстанция корректна

Рис. 2.15: Алгоритм проверки инстанции шаблона

Замечание. Ошибка *«не удалось вывести все параметры шаблона»* не является ошибкой в буквальном смысле. Она лишь свидетельствует о том, что не полностью указана конкретизация шаблона. Это может произойти, если не все параметры шаблона участвуют в типах аргументов (тогда их нельзя вывести по типам фактических аргументов при вызове), или при вызове указаны не все аргументы (так как имеет место частичное применение). Решить проблему можно явным указанием значений параметров шаблона. Пример приведен на листинге 2.1.

```
fun compose[!T, S, U](S -> U g, T -> S f, T x)
    return g(f(x));
end

Int -> Double f;
Double -> Bool g;

main
    compose(g);                    // ERROR
```

```
    compose[!S=Double](g);   // ERROR

    compose(g, f, 10);
    compose(g, f);
    compose[!T=Int](g);
end
```

Листинг 2.1: Пример полных и неполных конкретизаций шаблона при вызове

Можно использовать и другой подход — просто считать не полностью конкретизированный шаблон новым шаблоном функции с подстановкой типов $\gamma \circ \sigma$ и множеством ограничений $C_r'^{tp}$.

2.6 Вызов инстанций шаблонов в теле шаблона

Неосвещенным остался только вопрос вызова инстанции шаблона внутри другого шаблона.

Пусть на этапе построения ограничений встретился оператор

$$\texttt{temp[!T}_{i_1} = \texttt{U}_1, \ldots, \texttt{T}_{i_l} = \texttt{U}_l\texttt{]}(\texttt{V}_1, \ldots, \texttt{V}_m), 0 \leq l \leq k, 1 \leq m \leq n,$$

где `temp` — откомпилированный шаблон с теми же характеристиками, что и в предыдущем пункте 2.5 (об инстанцировании шаблона), а $\texttt{V}_i$ уже не обязаны быть конкретными типами. Алгоритм работы частично будет похож на алгоритм проверки инстанции на рис. 2.15.

Множество ограничений внешнего шаблона обозначено C.

Сначала проверим, что при вызове инстанции все параметры шаблона могут быть выведены (алгоритм на рис. 2.16).

обозначим $C^b = \{\mathtt{E}_{i_j} = \mathtt{U}_j\}_{j=1}^{l} \cup \{\mathtt{P}_j = \mathtt{V}_j\}_{j=1}^{m}$
если $\gamma = unify(C^b)$ завершилось ошибкой, то
  ошибка «некорректная инстанция, множество ограничений противоречиво»
иначе
  $C_r'^{tp} = \gamma C_r^{tp}$
  если применение подстановки завершилось ошибкой, то
    ошибка «некорректная инстанция, множество ограничений противоречиво»
  иначе
    если $\exists\ i, 1 \le i \le k \mid (\mathtt{E}_i = \gamma\mathtt{E}_i) \wedge (FV(\mathtt{E}_i) \neq \varnothing)$ (то есть типовые переменные выражения параметра не зависят от внешних типов), то
      ошибка «не удалось вывести все параметры шаблона»
    иначе успех

Рис. 2.16: Проверка инстанции

Если проверка прошла успешно, то просто внесем ограничения возможных типов внутреннего шаблона во множество ограничений C внешнего. Поскольку один и тот же шаблон может быть вызван неоднократно с разными типами, то для каждого вызова надо создать свою копию ограничений с новыми типовыми переменными. Через Ψ обозначим множество всех типовых переменных, которые встречаются в шаблоне `temp`. Запись $new\_tv()$ означает создание новой типовой переменной. Алгоритм приведен на рис. 2.17.

введем подстановку $\delta = \bigcup_{\mathtt{X} \in \Psi} [\gamma\mathtt{X} \mapsto new\_tv()]$
$nC_r^{tp} = \delta C_r'^{tp}$ (фактически это создание копии ограничений, ошибки не будет)
$C = C \cup nC_r^{tp}$

Рис. 2.17: Вызов инстанции шаблона в теле шаблона

Ранее мы рассматривали алгоритм анализа ограничений шаблона и считали, что после построения ограничений множество C состоит из базовых ограничений и ограничений применения. Оказалось, что если в теле шаблона содержится вызов шаблона, то C содержит также ограни-

чения возможных типов. Это не добавляет большой сложности. Нужно выделить из C подмножество ограничений данного вида C_0^{tp}. А затем на этапах, предшествующих этапу анализа этих ограничений, необходимо следить за C_0^{tp} и применять к нему подстановку типов, получаемую на каждом шаге.

Глава 3

Реализация

Практическая часть работы заключается в реализации **front-end и middle-end частей** компилятора модельного языка PollyTL. Это проект на языке C#, выполненный в среде Microsoft Visual Studio 2010 (платформа .NET Framework 4).

Проект включает около 30 файлов и содержит в общей сложности $\sim$ 18000 строк кода. Из них $\sim$ 4000 сгенерированы автоматически.

3.1 Front-end

Для реализации front-end компилятора было выбрано средство автоматической генерации лексических и синтаксических анализаторов `The Gardens Point Parser Generator (GPPG)` [8]. По `.lex` файлу с правилами лексического анализа и `.yacc` файлу с грамматикой языка GPPG генерирует программу-парсер на языке C#.

Таким образом задача данного этапа во многом заключается в составлении грамматики, соответствующей языку PollyTL. Грамматика приведена в приложении 1.

Результатом работы парсера является синтаксическое дерево программы. Поэтому необходимо также реализовать классы (соответствующие грамматике), экземпляры которых являются узлами этого дерева.

Приведем краткую информацию об основных файлах, соответствующих данной части проекта:

`polly.lex` правила лексического анализа;

`polly.y` грамматика языка;

`SSTreeNodes.cs` классы, соответствующие синтаксическому дереву программы (базовый класс `SyntaxSemanticTreeNode` и его производные, а также некоторые вспомогательные классы);

`PollyTLParserTools.cs` статический класс `PT`, занимающийся информированием о синтаксических и семантических ошибках;

`ErrorsManagment.cs` пространство имен, содержащее классы, связанными с обработкой ошибок (классы `SyntaxError` и `SemanticError` (производные от `Error`), а также обработчик информации об ошибках `ErrorsProcesser`).

Файлы `pollytl.cs`, `ShiftReduceParserCode.cs` и `pollytlyacc.cs` генерируются GPPG по файлам `polly.lex` и `polly.y` автоматически.

Для синтаксического анализа необходимо создать объект класса `PollyTLanguageParser` (наследник класса `GPPGParser`, файл `Parser.cs`) и применить метод `Parse`:

```
scanner = new Scanner();
scanner.SetSource(sourceCode, 0);
...
PollyTLanguageParser parser = new
    PollyTLanguageParser(scanner);
...
try { parser.Parse(); }
```

Если синтаксический анализ прошел успешно, то поле `parser.root` содержит ссылку на узел построенного синтаксического дерева.

3.2 Middle-end

Общая схема работы

Программа-парсер, полученная в результате работы генератора GPPG, проверяет только синтаксис входной программы. Проверкой семантики занимается middle-end.

Построенное парсером синтаксическое дерево на самом деле рассматривается как синтактико-семантическое. Проверка семантики реализуется проходом по этому дереву. В процессе обхода выполняется:

1. проверка и вывод типов выражений;
2. работа с таблицей символов (запись информации о функциях, переменных и шаблонах функций; поиск символов)
3. анализ шаблонов, построение и проверка ограничений;
4. и т. д.

Обход дерева осуществляют специальные классы-визиторы. Их несколько — для обхода узлов с «конкретным контекстом» и для обхода шаблонов. Начинает обход объект класса `MainFirstSemanticVisitor` (определенный в файле `SSTree_MainFirstSemanticVisitor.cs`):

```
MainFirstSemanticVisitor visitor = new
    MainFirstSemanticVisitor();
try
{
    visitor.Visit(parser.root as CodeProgramBlock);
    ...
```

В методе `Visit(node)` происходит обработка текущего узла: выполняются необходимые семантические проверки, и вызываются методы `Visit` для дочерних узлов.

Проверка семантики в основном реализуется методами текущего узла (то есть методами классов-наследников `SyntaxSemanticTreeNode`). Они вызываются в нужном порядке в методах `Visit`. Некоторые часто

используемые семантические действия реализованы как методы статического класса `ProgramSemantics` (файл `ProgramSemantics.cs`).

За контекст программы (таблицы символов и типов, некоторые флаги компиляции, стандартные типы и т. д.) отвечает статический класс `ProgramParsingContext` (файл `ProgramParsingContext.cs`). Классы таблиц символов и типов и связанных с ними объектов определены в файле `SymbolTable.cs`.

Интерфейс `ITypePrinter` (файл `ITypePrinter.cs`) представляет принтер типов — объект, который в процессе компиляции получает и накапливает информацию о типах выражений, интересующих пользователя (то есть указанных в директиве `#print_type`). После компиляции программы эта информация может быть получена с помощью метода `getOutput`:

```
string typesInfo =
    ProgramParsingContext.TemplateTypePrinter.getOutput()
    + ...
    + ProgramParsingContext.MainTypePrinter.getOutput();
```

Если при компиляции программы обнаружена ошибка, она может быть получена с помощью класса `PT`:

```
string errorsInfo = PT.ErrProcesser.GetErrors();
```

Обработка шаблонов

Если в процессе обхода дерева обнаружен узел, соответствующий описанию шаблона, то создается объект класса-визитора `TemplateBodyVisitor` (файл `SSTree_TemplateBodyVisitor.cs`):

```
public void Visit(TemplateFunctionDeclaration tempFunDef)
{
    ...
    TemplateBodyVisitor templVis = new
        TemplateBodyVisitor(tempFunDef);
```

Данный визитор занимается анализом операторов тела шаблона и построением ограничений. Классы ограничений определены в файле `Constraints.cs`.

Анализ построенного множества ограничений выполняет объект класса `ConstraintProcessor` (файл `ConstraintProcessor.cs`). Результатом является оставшееся множество ограничений и подстановка типов (объект класса `Substitution`, файл `Substitution.cs`).

```
ConstraintsProcessor constrProc = new ConstraintsProcessor(
    constraints, ...);
Substitution subs = constrProc.ProcessConstraints();
```

Процесс построения и анализа ограничений сопровождается логированием информации об ограничениях и подстановке типов. Это выполнят объект класса, реализующего интерфейс `IDebugInfoPrinter`.

Если анализ ограничений завершился успешно, то происходит применение подстановки типов к типам всех выражений шаблона. Применением подстановки занимается объект класса-визитора `TemplateSubsVisitor`.

В файле `TemplatesSupport.cs` содержатся вспомогательные классы, необходимые при работе с ограничениями.

3.3 Руководство пользователя

Проект представляет собой оконное приложение. Общий вид приведен на рис. 3.1.

В поле ввода слева вверху может быть набран код программы на языке PollyTL. Команда «открыть файл» (`Ctrl` + `O`) позволяет загрузить текст выбранного файла. Команда «сохранить файл» (`Ctrl` + `S`) сохраняет его на диск. Загружать файл в окно необязательно — можно поставить флажок «использовать файл» и выполнить команду «выбрать файл» (`Ctrl` + `L`).

При нажатии кнопки «компилировать» (`F5`) происходит синтакси-

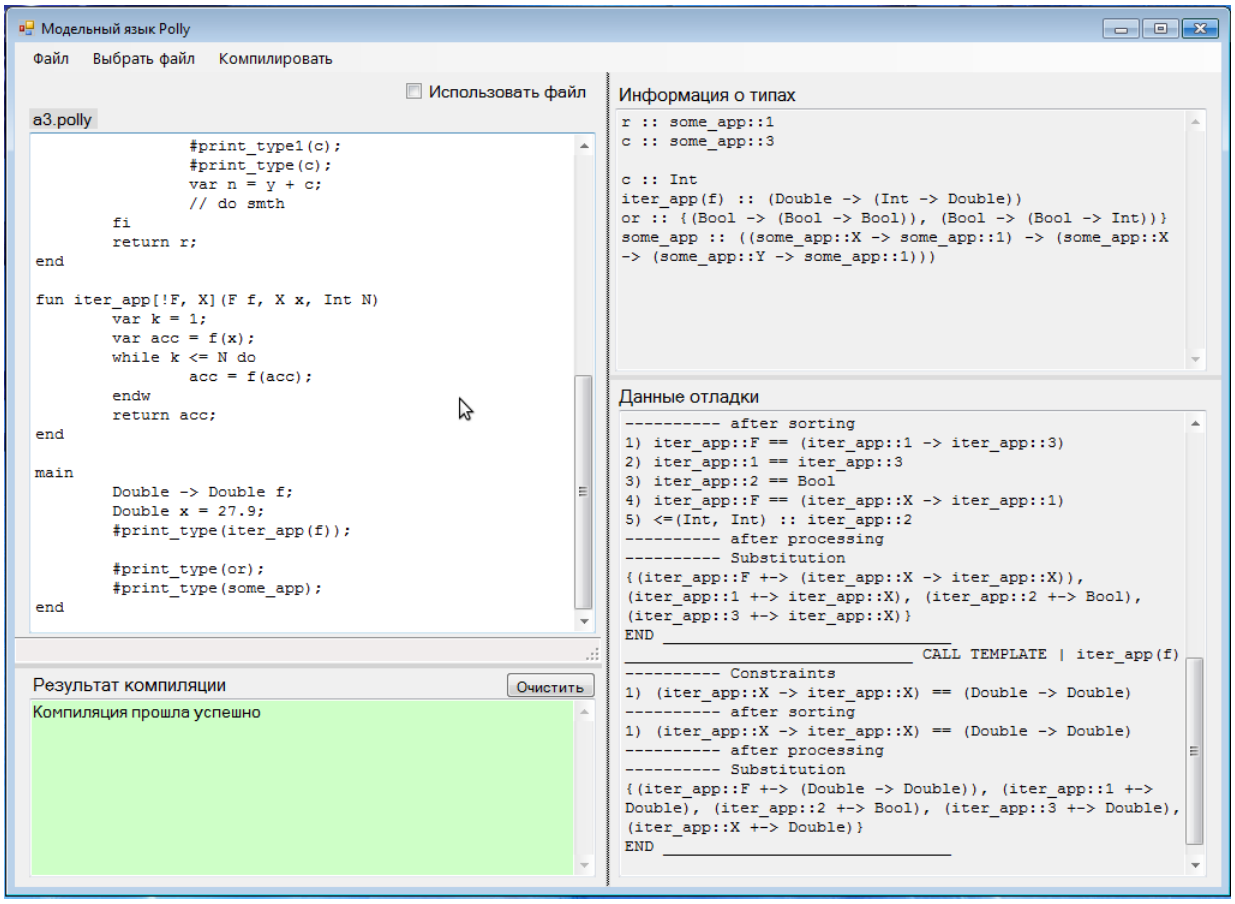

Рис. 3.1: Вид приложения

ческий и семантический анализ поданной программы, перевод во внутреннее представление. При этом в процессе анализа:

- в правое верхнее поле («информация о типах») может быть выведена информация о типах выражений (если в исходной программе присутствуют директивы компилятора `#print_type` или `#print_type1`);
- в правое нижнее поле («данные отладки») выводится информация о построенных ограничениях на шаблоны и результатах анализа шаблонов (то есть результирующий набор ограничений и подстановка типов), а также информация об инстанцировании шаблонов.

В поле «результат компиляции» помещается сообщение об успешной компиляции или текст ошибки.

3.4 Тестирование

Особенность тестирования состоит в том, что результат компиляции программы на модельном языке PollyTL не с чем сравнивать. Можно лишь провести анализ результатов вручную, основываясь на правилах и алгоритмах, рассмотренных ранее.

Далее мы рассмотрим несколько примеров программ с использованием шаблонов и проанализируем результаты компиляции. Будем приводить текст программы, а также полученную информацию о типах и данные анализа шаблонов.

Пример 1. Тождественная функция.

```
T fun id[!T](T x)
    return x;
end

main
    #print_type(id);
    #print_type(id(5));
end
```

Результат:

```
id :: (id:T -> id:T)
id(5) :: Int
```

```
id
Constraints: ∅
after processing: ∅
Substitution: []
id(5)
Constraints: {id:T == Int}
after processing: ∅
Substitution:[(id:T +-> Int)]
```

Функция `id` просто возвращает свой аргумент, ей все равно, какого он типа. При вызове `id(5)` тип аргумента `Int` сопоставляется с типом `T`.

Пример 2. Применение функции.

```
fun app1[!F, T](F f, T x)
    var r = f(x);
    #print_type(r);
    return r;
end
fun app2[!T, S](T -> S f, T x)
    var r = f(x);
    #print_type(r);
    return r;
end

Int fun succ(Int x)
    return x + 1;
end

main
    #print_type(app1);
    #print_type(app2);
    #print_type(app1(succ));
end
```

Результат:

```
r :: app1:1, r :: app2:S
app1 :: (app1:T -> app1:1) -> (app1:T -> app1:1)
app2 :: (app2:T -> app2:S) -> (app2:T -> app2:S)
app1(succ) :: Int -> Int
```

app1
Constraints: {app1:F == (app1:T -> app1:1)}
after processing: ∅
Substitution: [(app1:F +-> (app1:T -> app1:1))]
app2
∅
app1(succ)
Constraints: {(app1:T -> app1:1) == (Int -> Int)}
after processing: ∅
Substitution: [(app1:F +-> (Int -> Int)), (app1:1 +-> Int),
(app1:T +-> Int)]

С функцией app2 все просто. Функцию типа T → S можно применить к аргументу типа T, и результат будет иметь тип S. Об этом и говорит запись «r :: app2:S».

В случае с app1 изначально мы не знаем, что F — это функция. Но применение f(x) означает, что F на самом деле является функциональным типом, причем типом аргумента является T, а для возвращаемого значения нужно завести новую типовую переменную app1:1.

Пример 3. Применение с ограничением.

```
fun app[!F, T](F f, T x, T y)
    var a = f(x);
    var b = f(y);
    #print_type1(a);
    #print_type1(b);
    return a + b;
end

main
    #print_type(app);
end
```

Результат:

```
a :: app:1
b :: app:2
app :: (app:T -> app:1) -> (app:T -> (app:T -> app:3))
app
Constraints: {app:F == (app:T -> app:1), app:F == (app:T -> app:2),
+(app:1, app:2) :: app:3}
after processing: {(app:1, app:3) from {(Int, Int); (Double, Double)}}
Substitution: [(app:F +-> (app:T -> app:1)), (app:2 +-> app:1)]
```

На этапе построения ограничений типам `f(x)` и `f(y)` отвечают разные новые типовые переменные. Но поскольку в обоих случаях вызывается одна и та же функция типа `F`, то ясно, что `f(x)` и `f(y)` на самом деле они имеют один и тот же тип. Поэтому получена подстановка `app:2` $\mapsto$ `app:1`.

Функция + перегружена. Среди её версий есть только две с одинаковыми типами параметров: +(`Int`, `Int`) :: `Int` и +(`Double`, `Double`) :: `Double`. Отсюда получаем ограничение `(app:1, app:3) from {(Int, Int); (Double, Double)` (нетрудно догадаться, что типовая переменная `app:3` отвечает типу выражения `a + b`).

Пример 4.

```
fun abs[!T](T x)
    if x < 0 then
        return -x;
    fi
    return x;
end

fun expr[!F, T](F a, T x, T y)
    return abs(a) * abs(x) / y;
end

main
    #print_type(abs, abs(3), abs(-2.7));
    #print_type(expr, expr[!T=Double](3));
```

end

Результат (данные об инстанциях `abs(-2.7)` и `expr[!T=Double](3)` не приводятся):

```
abs :: (abs:T -> abs:T)
abs(3) :: Int
abs(-U(2,7)) :: Double
expr :: (expr:F -> (expr:T -> (expr:T -> Double)))
abs
Constraints: {<(abs:T, Int) :: abs:1, abs:1 == Bool, -U(abs:T) :: abs:2,
abs:2 == abs:T}
after processing: {(abs:T) from {(Int); (Double)}}
Substitution: [(abs:2 +-> abs:T), (abs:1 +-> Bool)]
expr
Constraints: {(expr:1/abs:T) from {(Int); (Double)},
expr:1/abs:T == expr:F, (expr:2/abs:T) from {(Int); (Double)},
expr:2/abs:T == expr:T, *(expr:1/abs:T, expr:2/abs:T) :: expr:1,
/(expr:1, expr:T) :: expr:2}
after processing: {(expr:T) from {(Int); (Double)},
(expr:F) from {(Int); (Double)},
(expr:1, expr:T) from {(Int, Int); (Double, Double); (Int, Double); (Double,
Int)},
(expr:F, expr:T, expr:1) from {(Int, Int, Int); (Double, Double, Double);
(Int, Double, Double); (Double, Int, Double)}}
Substitution: [(expr:1/abs:2 +-> expr:F), (expr:1/abs:1 +-> Bool),
(expr:2/abs:2 +-> expr:T), (expr:2/abs:1 +-> Bool), (expr:1/abs:T +->
expr:F), (expr:2/abs:T +-> expr:T), (expr:2 +-> Double)]
abs(3)
Constraints: {(abs:T) from {(Int); (Double)}, abs:T == Int}
after processing: ∅
Substitution: [(abs:2 +-> Int), (abs:1 +-> Bool), (abs:T +-> Int)]
```

Ограничение `<(abs:T, Int) :: abs:1` возникает в силу выражения `x < 0`, так как `0 :: Int`. А ограничение `abs:2 == abs:T` появляется благодаря тому, что возвращаемый тип функции один, а значит типы выражений `x` и `-x` в операторах `return` должны совпадать.

В шаблоне функции `expr` ограничения шаблона `abs` дублируются с новыми типовыми переменными для вызовов `abs(a)` и `abs(x)`. К примеру, типовая переменная `expr:2/abs:T` ранее не встречалась и соответствует параметру `T` внутреннего шаблона `abs`. Кроме того, добавляются ограничения `expr:1/abs:T == expr:F` и `expr:1/abs:T == expr:T`, характерные для инстанций. Операция $*$ допустима для четырех различных сочетаний типов `Int` и `Double` (ограничение `(expr:1, expr:T) from ...`), а вот результирующий тип операции / всегда равен `Double`. Отсюда подстановка `expr:2` $\mapsto$ `Double`.

Замечание №1. Аналогичного результата можно добиться, используя вместо шаблона `abs` перегруженную функцию с типами экземпляров `Int -> Int` и `Double` $\to$ `Double`.

Замечание №2. Вызовы `abs(true)` и `expr[!T=Bool](3)` корректно вызывают ошибку компиляции.

Замечание №3. На данном примере хорошо видны недостатки отложенного анализа ограничений возможных типов, о котором было сказано в п. 2.4.4. В частности, видна избыточность ограничений: присутствуют ограничения `(expr:1, expr:T) from ...` и `(expr:F, expr:T, expr:1) from ...`, хотя первое ограничение явно лишнее.

Пример 5. Неоднозначность применения перегруженной функции.

```
// or :: Bool -> (Bool -> Bool)
fun or(Bool a, Bool b)
    return a || b;
end
// or :: Bool -> (Bool -> Int)
fun or(Bool a, Bool b)
    if a || b then
        return 1;
    else
        return 0;
```

```
    fi
end
// or :: Int -> (Int -> Bool)
fun or(Int a, Int b)
    return (a > 0) || (b > 0);
end
// or :: Int -> (Int -> Int)
fun or(Int a, Int b)
    if or(a, b) then
        return 1;
    else
        return 0;
    fi
end

fun useOr[!T, F, S](T a, T b, F f, S x)
    var c = or(a, b);
    return f(x);
end

main
    useOr[!F=Double -> Double](-4, 5);
    useOr[!F=(Int -> Double) -> Bool](false, false);
end
```

Результат: *Ни при каких значениях типовых переменных нельзя вывести все типы шаблона useOr.*

useOr

Constraints: {or(useOr:T, useOr:T) :: useOr:1, useOr:F == (useOr:S -> useOr:2)}

after processing: {(useOr:T, useOr:1) from {(Bool, Bool); (Bool, Int); (Int, Bool); (Int, Int)}}

Substitution: [(useOr:F +-> (useOr:S -> useOr:2))]

Действительно, независимо от допустимого значения типовой переменной `T` (`Int` или `Bool`), остается по две подходящих версии перегруженной функции `or`.

А в следующем примере все будет работать правильно, так как для возвращаемого значения вызова `or` появится новое ограничение на равенство типу `Bool`.

```
fun useOr[!T, F, S](T a, T b, F f, S x)
    var c = or(a, b);
    if c then
        ; // do smth
    fi
    return f(x);
end

main
    useOr[!F=Double -> Double](-4, 5);
    useOr[!F=(Int -> Double) -> Bool](false, false);
end
```

Результат:

```
useOr
Constraints: {or(useOr:T, useOr:T) :: useOr:1, useOr:1 == Bool,
useOr:F == (useOr:S -> useOr:2)}
after processing: {(useOr:T) from {(Bool); (Int)}}
Substitution: [(useOr:F +-> (useOr:S -> useOr:2)), (useOr:1 +-> Bool)]
useOr[!F = Double -> Double](-4, 5)
Constraints: {(useOr:T) from {(Bool); (Int)}, useOr:T == Int,
(useOr:S -> useOr:2) == (Double -> Double)}
after processing: ∅
Substitution: [(useOr:F +-> (Double -> Double)), (useOr:1 +-> Bool),
(useOr:T +-> Int), (useOr:2 +-> Double), (useOr:S +-> Double)]
```

Заключение

Мы показали, что идея алгоритма реконструкции типов (`анализ выражений` $\Rightarrow$ `построение ограничений` $\Rightarrow$ `анализ ограничений`) может быть использована при реализации механизма шаблонов функций. Имеется в виду такой механизм, при котором:

1. на этапе компиляции шаблона проводится **максимальная проверка семантики** и **автоматическое построение ограничений** на параметры шаблона;
2. на этапе инстанцирования достаточно проверить типы на соответствие ограничениям.

Этот подход хорош тем, что позволяет получить **раннее обнаружение ошибок**, **быстрое инстанцирование** шаблона и **широкие возможности** по написанию обобщенного кода, при этом не требуя от пользователя дополнительных действий.

Недостаток заключается в том, что при компиляции шаблона, возможно, потребуется анализировать большую часть контекста.

В данной работе мы рассмотрели конкретную реализацию указанного подхода для модельного языка PollyTL. Таким образом, было сделано следующее:

1. Разработан модельный язык программирования с шаблонами функций и построена его грамматика.
2. Реализован front-end компилятор.
3. Разработан механизм шаблонов, основанный на автоматическом

построении ограничений, с ранним обнаружением ошибок компиляции и быстрым инстанцированием.

4. Построены правила анализа тела шаблона.
5. Сконструированы алгоритмы построения, преобразования и анализа ограничений.
6. Реализован middle-end компилятор модельного языка, включающий указанный механизм шаблонов.

Литература

[1] *David R. Musser and Alex Stepanov*. Generic programming. In ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation, 1988.

[2] *Alexander A. Stepanov and Meng Lee*. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[3] *Универсальные шаблоны .NET* `http://msdn.microsoft.com/ru-ru/library/ms172192%28v=vs.90%29.aspx`.

[4] *Бенджамин Пирс.* Типы в языках программирования. М.: Изд-во ДМК, 2012.

[5] *Миран Липовача.* Изучай Haskell во имя добра!. М.: Изд-во «Лямбда пресс» & «Добросвет», 2012.

[6] *Jeremy G. Siek*. A Language for Generic Programming. Indiana University, August 2012.

[7] *Альфред Ахо, Рави Сети, Джеффри Ульман.* Компиляторы. Принципы, технологии, инструменты. М.: Изд-во Вильямс, 2001.

[8] *The Gardens Point Parser Generator (GPPG)* `http://plas.fit.qut.edu.au/gppg/`

Приложение 1. Грамматика языка Polly

Основные блоки

```
codefile                                    // Кодовый файл
    : ProgramBlock

ProgramBlock                                // Программный блок
    : Declarations mainProgramFunc

mainProgramFunc                             // Основной исполняемый раздел
    : main Statements end
```

Директива компилятора

```
CallFuncDebugDirective
    : # ident ( FunctionFactParameters )
```

Типы

```
TypeDescription                             // Описание типа
    : StadardType
    | ArrowType

StadardType                                 // Обычный именованный тип
    : ident

ArrowType                                   // Функциональный тип
    : InArrowType -> ArrowType
    | InArrowType -> InArrowType
```

```
InArrowType                          // Часть функционального типа
    : StadardType
    | ( ArrowType )
```

Описания

```
Declarations                         // Раздел описаний
    : DeclarationsList
    | пусто

DeclarationsList                     // Список секций описаний
    : DeclarationsList DeclarationSection
    | DeclarationSection

DeclarationSection                   // Секция описания
    : FunctionDeclarationSection
    | VariableDeclarationSection

VariableDeclarationSection           // Описание переменных
    : AutoTypeInferenceVarDecSection
    | TypeDescription VariableDefinitionsList ;

AutoTypeInferenceVarDecSection       // Оп. перем. с автовыводом типа
    : var ident = expr ;

VariableDefinitionsList              // Список описаний переменных
    : VariableDefinitionsList , VariableDefinition
    | VariableDefinition

VariableDefinition                   // Описание переменной
    : ident
    | ident = expr


FunctionDeclarationSection           // Описание функции или шаблона
    : ReturnType fun ident TemplateParams
```

```
        ( FunctionFormalParameters ) Statements end
    | ReturnType FUN ident
        ( FunctionFormalParameters ) Statements end

ReturnType                              // Возвращаемый тип функции
    : TypeDescription
    | пусто

FunctionFormalParameters                // Формальные параметры функции
    : FormalParametersList
    | ParameterDeclaration
    | пусто

TemplateParams                          // Параметры шаблона функции
    : [! IdentList ]

FormalParametersList                    // Список формальных параметров
    : FormalParametersList , ParameterDeclaration
    | ParameterDeclaration , ParameterDeclaration

ParameterDeclaration                    // Объявление параметра
    : TypeWithIdentParameter
    | ident

IdentList                               // Список идентификаторов
    : IdentList , ident
    | ident

TypeWithIdentParameter                  // Некоторый параметр вида тип + идентиф
    : TypeDescription ident
```

Выражения

```
ident                                   // Идентификатор
    : ID

expr                                    // Выражение
```

```
    : expr Relation SimpleExpr
    | SimpleExpr

Relation                                // Отношение
    : == | !=
    | >= | <= | > | <

SimpleExpr                              // Простое выражение
    : SimpleExpr PlusOperator SignedTerm
    | SignedTerm

PlusOperator                            // Оператор сложения
    : + | - | <||>

SignedTerm                              // Слагаемое со знаком
    : term
    | + term %prec UPLUS
    | - term %prec UMINUS

term                                    // Слагаемое
    : term MultOperator factor
    | factor

MultOperator                            // Оператор умножения
    : * | / | &&
    | div | mod

factor                                  // Множитель
    : ident
    | ! factor
    | BoolValue
    | NumericValue
    | CallFunction
    | ExplicitTemplateCallFunction
    | LambdaExpr
    | ( expr )
```

```
BoolValue                              // Булево значение
    : true | false

NumericValue                           // Числовое выражение
    : INTNUM
    | DOUBLENUM


LambdaExpr                             // Лямбда — выражение
    : LambdaParameters => { LambdaBody }

LambdaParameters                       // Аргументы лямбда — выражения
    : ident
    | TypeWithIdentParameter
    | ( FormalParametersList )

LambdaBody                             // Тело лямбда — выражения
    : expr
    | StatementsList

CallFunction                           // Вызов функции
    : ident ( FunctionFactParameters )

FunctionFactParameters                 // Фактические параметры функции
    : ExprList
    | пусто

ExplicitTemplateCallFunction           // Вызов шаблона функции
    : ident [! TemplateTypesList ] ( FunctionFactParameters )

TemplateTypesList                      // Список указаний параметров шаблона
    : TemplateTypesList , TemplateTypeDeclaration
    | TemplateTypeDeclaration

TemplateTypeDeclaration                // Указание типа шаблона
```

```
    : TypeDescription
    | ident = TypeDescription

ExprList                                // Список выражений
    : ExprList , expr
    | expr
```

Замечание. Видно, что грамматика включает конструкции для описания лямбда-выражений (лямбда-функций). На уровне семантики они не были реализованы, поэтому в главе 1 о них не упоминается.

Операторы

```
Statements                              // Раздел операторов
    : StatementsList
    | пусто

StatementsList                          // Последовательность операторов
    : StatementsList Statement
    | Statement

Statement                               // Оператор
    : InternalDeclarations
    | Assignment
    | IfStatement
    | WhileStatement
    | EmptyStatement
    | CallFunction ;
    | ExplicitTemplateCallFunction ;
    | ReturnOperator
    | CallFuncDebugDirective

EmptyStatement                          // Пустой оператор
    : SEMICOLUMN ;

InternalDeclarations                    // Описания, возможные внутри программы
    : VariableDeclarationSection
```

```
Assignment                              // Присваивание
    : ident = expr ;

IfStatement                             // Оператор If
    : if expr then StatementsList ElifIfStatementPart fi
    | if expr then StatementsList ElifIfStatementPart
        else StatementsList fi

ElifIfStatementPart                     // Набор Elif – операторов
    : ElifStatementsList
    | пусто

ElifStatementsList                      // Список Elif – операторов
    : ElifStatementsList ElifStatement
    | ElifStatement

ElifStatement                           // Elif – выражение
    : elif expr then StatementsList

WhileStatement                          // Оператор while
    : while expr do StatementsList endw

ReturnOperator                          // Возврат значения
    : return ;
    | return expr ;
```