УДК 004.432.4

Анализ типов в трансляторе с языка предикатного программирования

Зубарев А.Ю. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Семантика языка предикатного программирования P формализована с использованием трех видов отношений: совместимости, согласованности и тождества. Рекурсивные типы определены через аппарат наименьшей неподвижной точки. Обобщенные типы представлены типовыми ограничениями (концептами). Для конструкций с неявной типизацией сформулированы правила восстановления типов переменных. Разработаны алгоритмы проверки корректности рекурсии, определения типов для языковых конструкций, проверки семантической корректности конструкций.

Ключевые слова: *тип данных, семантический анализ, доказательное программирование, предикатное программирование, обобщенное программирование.*

1. Введение

Язык предикатного программирования P [2] является языком доказательного программирования со статической типизацией. В языках доказательного программирования программа задается вместе с ее спецификацией: предусловиями и постусловиями для всех подпрограмм. Поскольку почти все системы автоматического доказательства базируются на тотальных функциях, типы данных должны быть точно определены. В системе типов становится необходимым использовать *подтип* как множество истинности некоторого предиката. Следствием этого является параметризация типов переменными.

В языках доказательного программирования нет указателей. Вместо них используются рекурсивно определяемые алгебраические типы; во многих языках определяемые конструкцией **datatype**.

Другая особенность – использование произвольных типов как параметров, где тип представлен только именем. Параметризация типами характерна для *обобщенного программирования*, обеспечивающего повторную используемость программных компонент за счёт отделения реализации алгоритмов от конкретных определений типов данных [9].

В языке предикатного программирования P имеются конструкции с *неявной типизацией* – типы переменных в таких конструкциях явно не заданы и при трансляции восстанавливаются из контекста.

Анализ типов с учетом указанных особенностей системы типов становится принципиально сложнее, чем для языков императивного программирования.

Обзор работ. Анализ типов является популярной проблематикой в теории трансляции. В работах [11], [7] описаны основные концепции системы типов разных языков программирования. Объект данных определяется как четверка (L,N,V,T), где L – место нахождения объекта, N – его имя, V – его значение и T – тип объекта. Проверка типов — это процесс определения типа указанного объекта данных. Язык программирования является строго типизированным, если проверка типов происходит во время компиляции. Одной из затрагиваемых проблем в работе [7] является способы преобразования объектов данных одного типа в другой. Выделяют две стратегии преобразования типов – явное и неявное (приведение типов). Рассматриваются два возможных определения эквивалентности типов:

1. Два объекта данных имеют эквивалентный тип, если их типы описывают одно множество значений (структурная эквивалентность).

2. Два объекта данных имеют эквивалентный тип, если они имеют одно имя (именная эквивалентность).

Отношение вложенности типов и основанные на нем правила семантики языков анализируются в работе [10]. Параметризация типов в контексте обобщённого программирования рассматривается в работе [4]. В C++ свободу использования обобщенного программирования дают неограниченные шаблоны. Однако при их использовании возникает ряд проблем, таких как сложность поиска и исправления ошибок, недоступность раздельной трансляции шаблонов классов и функций. Отличие подходов C++ и Java/C# следующее: в шаблонах C++ разрешено все, что не запрещено, а в обобщенных типах Java/C#, наоборот, запрещено все, что не разрешено. В языке Scala присутствуют абстрактные типы-члены. Такие типы могут быть ограничены, при этом существует несколько способов описания этих ограничений. Ограничения на типы используются компилятором для доказательства того, что код удовлетворяет системе типов.

Целью данной работы является разработка алгоритмов анализа типов предикатной программы как части семантического анализа в экспериментальной системе предикатного программирования.

В разделе 2 описывается синтаксис языка P, связный с описанием типовых термов, их параметризации и заданием имени типа. Рассматриваются конструкции языка, описывающие

эквивалентные типы данных. *В разделе 3* описывается понятие языковой конструкции, ее подконструкций и позиций. Вводятся вспомогательные термины и обозначения. *В разделе 4* описаны рекурсивные типы данных. Вводятся ограничения на рекурсивные типы, с использованием аппарата неподвижной точки, доказывается их корректность. *В разделе 5* вводится понятие вложенности типов; на его основе строятся три вида отношений: согласованность, совместимость и тождество типов. *В разделе 6* формализуются ограничения на типы языковых конструкций в терминах трех отношений. *В разделе 7* рассматривается параметризация типов и вводится понятия обобщенных типов. Вводятся ограничения (концепты) для таких типов. В *разделе* 8 рассматриваются конструкции с неявной типизацией, приводятся правила восстановления типов для этих конструкций. *В разделе 9* описываются этапы семантического анализа предикатной программы. *В разделе 10* сформулированы основные результаты работы.

2. Синтаксис типовых термов

Предикатная программа определяет предикат в форме вычислимого оператора. Полная предикатная программа языка P состоит из набора рекурсивных предикатных программ, которые определяются следующей конструкцией:

A **(**x**:** y**) pre** P**(**x**) post** Q**(**x**,** y**) {** S **};**

где A – имя предиката, x и y – непересекающиеся наборы переменных (аргументы и результаты), P и Q – логические формулы (предусловие и постусловие), S – оператор.

В экспериментальной системе предикатного программирования программа транслируется во внутреннее представление, из которого генерируется код на языках PVS и C++.

Типы языка P являются примитивными или составными. *Примитивные типы*: натуральный (**nat**), целый (**int**), вещественный (**real**), символьный (**char**) и логический (**bool**). *Составные типы* строятся на базе других типов. Синтаксически различные конструкции языка могут описывать эквивалентные типы.

Произвольный тип в предикатной программе определяется *типовым термом*, представляемым далее конструкцией <TERM>. Задание имени типа для типового терма осуществляется *описанием типа* в виде конструкции <DT>.

Синтаксис языковых конструкций будет описываться на расширенном языке Бэкусовских нормальных форм (БНФ) [8] со следующими особенностями:

- **Жирным** шрифтом выделены терминальные символы
- [*фрагмент*] - определяет возможное отсутствие фрагмента

- (*фрагмент*)* - определяет повторение фрагмента нуль или более раз; круглые скобки могут быть опущены, если фрагмент состоит из одного символа
- (*фрагмент*)+ - определяет повторение фрагмента один или более раз
- (*фрагмент*)^ - определяет список вида: *фрагмент* (**,** *фрагмент*)*

Ниже приведено описание синтаксиса конструкций языка P описывающих тип.

Описание типа:

DT ::= **type** IT [**(**PARAM**)**] = TERM | **type** IT

Параметры типа:

PARAM ::= TERM ID^ [, PARAM] | **type** ID^ [, PARAM]

где ID - *идентификатор*

Типовой терм:

```
TERM ::= PRIM |
         [ID .] IT [(CARG^)] |
         EXPR..EXPR |
         subtype (TERM ID: EXPR | var ID: EXPR) |
         predicate DP |
         struct ((TERM ID^)^) |
         union ((ID [(TERM ID^)^])^) |
         enum (ID^) |
         set (TERM) |
         array (TERM , TERM (, TERM)*) |
         class [extends ID] { DC (; DC)* } |
         list (TERM) |
         string
```

где IT *– имя типа,* DC *– описание класса,* EXPR *– выражение*

Примитивный тип:

PRIM ::= **nat** [DIGI] |**int** [DIGI] |**real** [DIG]| **bool** |**char**

Разрядность:

DIGI ::= 1 | 2 | ... | 64

DIG ::= 32 | 64 | 128

Предикатная программа:

```
DP ::= ((ARG]:RES) [pre F][post F]) |
       ((ARG]:[RES][#M](: [RES][#M])+)[pre F] [(pre M : F)*] [(post M : F)*]
```

где M *– метка*

Аргументы и результаты:

ARG ::= TERM ID[‘] (**,** ID[‘])* [**,** ARG] | **type** ID [**,** ARG]

RES ::= TERM ID[‘] (, ID[‘])* [**,** RES]

Аргумент вызова:

CARG := EXPR | TERM

Формула:

F ::= EXPR | (F) | ID**(**EXPR^**)** | **!**F | F **&** F | F **or** F | F **=>** F | F **<=>** F |
(Q (TERM ID)^)+ . F

Q ::= **forall** | **exists**

Множеством значений типа **subtype** является подмножество основного указываемого типа, для которого выполняется некоторое логическое выражение. В частности, тип натуральных чисел определяется следующим образом: **subtype** (**int** i: i>=0).

Значением типа **union** является значение одного из конструкторов, перечисленных в списке определения **union**. Конструктор определяется именем конструктора и набором полей.

Изображение отдельных типов языка P представлено в особом синтаксисе. Они определяются через базисные типовые термы следующим образом:

- Тип диапазона a**..**b эквивалентен типу **subtype** (T i: i>=a & i<=b).
- Тип **enum**(A) эквивалентен типу **union**(A).
- Тип **list**(T) вводится определением:

  type list(T) = **union** (**nil**, **cons** (T **car** , **list** (T) **cdr**)).
- Тип **string** эквивалентен типу **list**(**char**)
- Типы **union** с разными порядками одинаковых конструкторов следует считать эквивалентными.

3. Иерархия конструкций в языке P

Языковая конструкция – независимая часть программы. Синтаксис и семантика языка определяют множество допустимых текстов данной конструкции. Например, оператор и выражение – это конструкции. Тип может быть атрибутом конструкции.

Конструкция обычно составляется из *подконструкций*. Место подконструкции в составе объемлющей конструкции называется *позицией* подконструкции. Позиция может иметь ограничения на тип соответствующей ей конструкции. В простейшем случае позиция допускает определенный тип конструкций. Например, условный оператор имеет три

позиции. Позиция условия допускает только логический тип конструкции. Если конструкция составлена из нескольких подконструкций, то их позиции являются *соседними*.

Обозначение A<B> означает, что B является подконструкцией конструкции A. Разумеется, конструкция A может содержать другие подконструкции, но мы выделяем только B. Обозначение A[B] означает, что конструкция B *входит* в конструкцию A, т.е. A[B] эквивалентно $A<B> \vee \exists C.\ A<C>\ \&\ C[B]$; иначе говоря, B является частью A и находится на некотором уровне иерархии в структуре конструкции A. Обозначение $A[B_1...B_n]$ означает, что конструкции $B_1...B_n$ входят в конструкцию A.

Пусть S[x] некоторая конструкция, тогда под S[X] будем подразумевать конструкцию S[x], где x = X; под $S^2[X]$ конструкцию S[x], где x = S[X]; под $S^k[X]$ конструкцию S[x], где $x = S^{k-1}[X]$; под $S^0[X]$ конструкцию X.

4. Рекурсивные типы

Совокупность определений типов вида **type** IT [**(**PARAM**)**] = TERM может оказаться рекурсивной. Тип A *непосредственно зависит* от типа B при наличии определения вида **type** A = T[B], где T некоторый составной тип. A *определяется* через тип B, если существуют типы $X_1,...,X_m$ ($m > 1$) такие, что $A = X_1$, $B = X_m$ и X_i непосредственно зависит от X_{i+1} для $i = 1...m-1$. Тип B является *рекурсивным*, если B определяется через B.

Рекурсивным кольцом типа A является совокупность типов B, таких что A определяется через B и B определяется через A. Позиция в определении типа A называется *рекурсивной,* если в этой позиции находится рекурсивный тип и этот тип принадлежит рекурсивному кольцу типа A.

Корректность рекурсивных определений типов может быть обеспеченна аппаратом наименьшей неподвижной точки, который рассматривается в работе [5].

Набор определений типов рекурсивного кольца $X_1,...,X_N$ можно представить в виде:

type $X_k = f_k[X_1...X_N]$; $k = 1..n$,

где f_k – типовой терм. Отношение включения $\subseteq$ определяет нижнюю полурешетку на типах, описывающих множества данных, с пустым типом, обозначаемым как $\emptyset$. Перепишем систему в векторной форме:

$X = f[X]$,

где $X=(X_1...X_n)$ – вектор типов, $f=(f_1...f_n)$ вектор типовых термов.

Введем отношение $\sqsubseteq$ на типовых векторах: $X \sqsubseteq Y \cong \forall k=1..n\ (X_k \subseteq Y_k)$. Вектор $\Theta = (\emptyset, \emptyset,...,\emptyset)$ является минимальным элементом полурешетки определяемой отношением $\sqsubseteq$.

Пусть $(D, \sqsubseteq, \bot)$ полная решетка с наименьшим элементом $\bot$. Приведем некоторые определения математических понятий, используемых в дальнейшем изложении.

Последовательность $\{a_m\}_{m\geq 0}$ является *возрастающей цепью*, если $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_m \sqsubseteq \ldots$. Для наименьшей верхней грани цепи $\{a_m\}_{m\geq 0}$ будем использовать обозначение: $\cup_{m\geq 0} a_m$.

Лемма. Пусть $\{A_m\}_{m\geq 0}$ – возрастающая цепь, $A^{\sim} = \cup_{m\geq 0} A_m$. Пусть $a \in A^{\sim}$, тогда $\exists k\ \forall m \geq k.\ a \in A^m$.

Тотальная функция $F{:}D \to D$ называется *непрерывной*, если для любой возрастающей цепи $\{a_m\}_{m\geq 0}$ выполняется равенство $F(\cup_{m\geq 0} a_m) = \cup_{m\geq 0} F(a_m)$.

Неподвижной точкой функции F называется решение уравнения $x = F(x)$.

Пусть F произвольная функция, для натурального n определим $F_0(x)=x$, $F_{n+1}(x)=F_n(F(x))$

Лемма. $\{F_n(\bot)\}_{n\geq 0}$ для непрерывной функции F определяет возрастающую цепь.

Теорема Клини. Пусть F – непрерывная функция. Тогда $\cup_{n\geq 0}\{F_n(\bot)\}$ является наименьшей неподвижной точкой F. [3]

Рассмотрим последовательность типовых векторов $\{X^m\}_{m\geq 0}$ определяемую рекуррентно следующим образом: $X^0 = \Theta$, $X^{m+1} = f[X^m]$, $m \geq 0$. Естественно ожидать, что предел последовательности $\{X^m\}_{m\geq 0}$ (если он существует) даст нам неподвижную точку – решение системы $X = f[X]$.

Лемма. Вектор-функция f системы $X = f[X]$ определений рекурсивных типов является непрерывной относительно **struct** и **union**.

Доказательство.

Пусть типовой терм f имеет вид **struct (**$X_1\ g_1$, $X_2\ g_2$, ..., $X_n\ g_n$**)**, элементами соответствующего ему типа будем считать кортежи вида $(x_1 \ldots x_n)$, где x_i элемент множества X_i. Докажем, что f непрерывна относительно типов $X_1, X_2, \ldots, X_n$.

Требуется доказать, что

$\cup_{m\geq 0}$(**struct (**$X_1^m\ g_1$, ..., $X_n^m\ g_n$**)**) = **struct (**$\cup_{m\geq 0}(X_1^m)\ g_1$, ..., $\cup_{m\geq 0}(X_n^m)\ g_n$**)**

Докажем сначала, что тип левой части равенства содержится в типе правой части. Пусть $(x_1 \ldots x_n) \in \cup_{m\geq 0}$(**struct (**$X_1^m\ g_1$, ..., $X_n^m\ g_n$**)**). Тогда, согласно рассмотренной лемме, существует такое k, что $(x_1 \ldots x_n) \in$ **struct (**$X_1^m\ g_1$, ..., $X_n^m\ g_n$**)** для всех $m \geq k$. Далее, $x_i \in \cup_{m\geq 0} X_i^m$ и, следовательно, $(x_1 \ldots x_n) \in$ **struct (**$\cup_{m\geq 0}(X_1^m)\ g_1$, ..., $\cup_{m\geq 0}(X_n^m)\ g_n$**)**.

Допустим теперь, что $(x_1 \ldots x_n) \in$ **struct (**$\cup_{m\geq 0}(X_1^m)\ g_1$, ..., $\cup_{m\geq 0}(X_n^m)\ g_n$**)** и $x_i \in \cup_{m\geq 0} X_i^m$. Существуют такие k_i, что $x_i \in X_i^m$ для $m \geq k_i$. Пусть $k = \max(k_1,\ldots,k_n)$, тогда $(x_1 \ldots x_n) \in$ **struct (**$X_1^m\ g_1$, ..., $X_n^m g_n$**)** для $m \geq k$ и, следовательно, $(x_1 \ldots x_n) \in \cup_{m\geq 0}$(**struct (**$X_1^m\ g_1$, ..., $X_n^m\ g_n$**)**).

Пусть типовой терм f имеет вид **union (**C_1, C_2, ..., C_n**)**. Необходимо доказать, что терм f непрерывен относительно типов полей конструктора X_1, X_2, ..., X_m. непрерывность конструкторов C_1, C_2, ..., C_n относительно типов X_1, X_2, ..., X_m доказывается аналогично непрерывности **struct**. Докажем, что терм f непрерывен относительно конструкторов C_1, C_2, ..., C_n.

Требуется доказать, что

$\cup_{m\geq 0}$(**union (**C_1^m, ..., C_n^m**)**) = **union (**$\cup_{m\geq 0}(C_1^m)$, ..., $\cup_{m\geq 0}(C_n^m)$**)**

Пусть $x \in \cup_{m\geq 0}$(**union (**C_1^m, ..., C_n^m**)**). Тогда существует такое k, что $x \in$ (**union (**C_1^m, ..., C_n^m**)**) для всех $m \geq k$. Так как, значением типа **union** является значение одного из конструкторов, то $x \in \cup_{m\geq 0} C_i^m$ и, следовательно, $x \in$ **union (**$\cup_{m\geq 0}(C_1)$, ..., $\cup_{m\geq 0}(C_n)$**)**.

Допустим теперь, что $x \in$ **union (**$\cup_{m\geq 0}(C_1)$, ..., $\cup_{m\geq 0}(C_n)$**)** и $x \in \cup_{m\geq 0} C_i^m$. Существует такое k, что $x \in C_i^m$ для $m \geq k$, тогда $x \in$ **union (**C_1, ..., C_n**)** для $m \geq k$ и, следовательно, $x \in \cup_{m\geq 0}$(**union (**C_1, ..., C_n**)**). □

В соответствии с теоремой Клини о неподвижной точке решением системы $X = f[X]$ является неподвижная точка функции f.

Для построения списков и деревьев достаточно рекурсии с использованием **union** и **struct**. Возможны другие, экзотические формы рекурсии, однако они бесполезны при разработке реальных алгоритмов.

Пусть **type** A(x) = T[x] рекурсивный тип. Позиция K терма T *допускает рекурсию*, если выполнено одно из следующих условий:

- T является структурой и K является позицией типа поля;
- T является объединением и K является позицией типа поля конструктора;
- K является позицией типа поля конструктора объединения, стоящего в допускающей рекурсию позиции;
- K является позицией типа поля структуры, стоящей в допускающей рекурсию позиции.

Рекурсивная позиция должна быть позицией, допускающей рекурсию. Для непустого решения рекурсивного уравнения необходимо присутствие в рекурсивном кольце типа **union** с конструктором, не имеющим рекурсивных позиций.

5. Отношения на типах

Прежде чем определить отношение на типах введем вспомогательные обозначения и понятия.

Позицию K типа T будем называть *ковариантной,* если выполнено одно из следующих условий:

- T является структурным типом и K – позиция типа поля;
- T является типом объединения и K – позиция типа поля конструктора;
- T является типом массива и K – позиция типа элементов;
- T является типом множества подмножеств и K позиция базисного типа;
- K является ковариантной позицией некоторого типа, стоящего в ковариантной позиции типа T.

Последнее условие означает, что в иерархически определяемом типе все подуровни вверх от ковариантной позиции должны быть ковариантны.

Определим *предпорядок* на типовых термах. Бинарное отношение вложенности типов <: определяет подмножество декартова произведения TERM×TERM. Если $(t_1,t_2)\in$ <: то тип t_1 не превосходит тип t_2; в этом случае будем использовать традиционную запись вида t_1 <: t_2. Если t_1 <: t_2 и t_2 <: t_1, то будем писать $t_1 \sim t_2$.

По определению предпорядка это отношение должно удовлетворять следующим условиям:

- Рефлексивность: $\forall t.\ t <: t$
- Транзитивность: $\forall t_1,t_2,t_3.\ t_1 <: t_2\ \&\ t_2 <: t_3 \Rightarrow t_1 <: t_3$

Для системы типов языка P определим отношение вложенности типов <: следующим набором правил:

Примитивные типы:

1. **int** <: **real**
2. **nat** <: **int**
3. **int** ~ **int 32**
4. **real** ~ **real 64**
5. **nat** ~ **nat 32**
6. **int** d_1 <: **int** d_2, если $d_1 \le d_2$
7. **nat** d_1 <: **int** d_2, если $d_1 + 1 \le d_2$
8. **int** d_1 <: **real** 32, если $d_1 \le 24$
9. **int** d_1 <: **real** 64, если $d_1 \le 53$
10. **int** d_1 <: **real** 128, $\forall\ d_1$
11. **nat** d_1 <: **nat** d_2, если $d_1 \le d_2$

12. **real** d_1 <: **real** d_2, если $d_1 \le d_2$

Составные типы:

13. Если P_1, P_2 – логические выражения и $P_1(x) \Rightarrow P_2(x)$ $\forall x \in T_1$, $T_1 <: T_2$, тогда **subtype**(T_1 x: $P_1(x)$) <: **subtype**(T_2 x: $P_2(x)$).
14. **subtype**(T x: P(x)) <: T $\forall T \in TERM$.
15. T ~ **subtype**(T x: **true**) $\forall T \in TERM$.
16. Пусть m параметр, типа T_1. Если P_1, P_2 – логические выражения, $P_1(x,m) \Rightarrow P_2(x,m)$ $\forall m$ ($m \in T_1$, $x \in T_2$) и $T_2 <: T_3$, тогда **subtype**(T_2 x: P_1(x, m)) <: **subtype**(T_3 x: P_2(x,m)).
17. Пусть type A = **class** {…}, type B = **class** extends A {…}, тогда A <: B.
18. Тип **predicate** ((A_1: R_1) pre S_1 post F_1) <: **predicate** ((A_2: R_2) pre S_2 post F_2), если количество аргументов и результатов совпадают, типы из R_1 тождественны типам из R_2, типы из A_1 не превосходят соответствующие типы из A_2, $S_2 \Rightarrow S_1$ и $F_1 \Rightarrow F_2$.
19. Пусть T_1 = **struct** (...) и T_2 получен из T_1 перестановкой полей, тогда $T_1 \sim T_2$.
20. Пусть T_1 = **struct** (...) и T_2 получен из T_1 добавлением новых полей, тогда $T_1 <: T_2$.
21. Пусть типовой терм x находятся в ковариантной позиции типового терма T[x]. Тогда если X <: Y, то T[X] <: T[Y].

Типы по имени и рекурсивные типы.

22. Пусть **type** A_1**(x)** = T_1[x] и **type** A_2**(y)** = T_2[y]. Тогда если $T_1[X] <: T_2[Y]$, то $A_1(X) <: A_2(Y)$.
23. Пусть **type** A_1**(x)** = S_1[x] и **type** A_2**(y)** = S_2[y]. $T_1[A_1(X)] <: T_2[A_2(Y)]$, если $T_1[S_1[X]] <: T_2[S_2[Y]]$
24. Пусть **type** A**(x)** = T_1 [x], где T_2 некоторый типовой терм. Тогда если $T_1[X] <: T_2$, то $A(X) <: T_2$, а если $T_2 <: T_1[X]$, то $T_2 <: A(X)$.

    Пусть даны два рекурсивных типа **type** A_1**(x)** = T_1[x] и **type** A_2**(y)** = T_2[y], обозначение A_1**(X)** $\sqsubset:$ A_2**(Y)** означает, что:

    a) типы совпадают за возможным исключением ковариантных позиций
    b) типы, стоящие в нерекурсивных ковариантных позициях T_1[X], меньше либо равны соответствующим типам из T_2[Y].
25. Пусть **type** A_1**(x)** = T[x, A_1(S[x])] и **type** A_2**(y)** = T[y, A_2(L[y])] два рекурсивных типа. Если $\forall k \ge 0$. A_1**(**S^k[X]**)** $\sqsubset:$ A_2**(**L^k[Y]**)**, то $A_1(X) <: A_2(Y)$.

Заметим, что если x = S[x] и y = L[y], то из A_1**(**S[X]**)** $\sqsubset:$ A_2**(**L[Y]**)** следует $A_1(X) <: A_2(Y)$. Иначе ограничимся следующими случаями:

- На место параметра типа или параметра переменной в рекурсивном вызове было поставлено значение или типовой терм не зависящий от параметров, тогда из A_1**(**X**)** ⊏: A_2**(**Y**)** и A_1**(**S[X]**)** ⊏: A_2**(**L[Y]**)** следует $A_1(X) <: A_2(Y)$.
- В S имеет место преобразование параметра переменного (n ⇒ f(n)). Все типы **subtype** из T_1, в которых используется этот параметр, сравниваются с типами из T_2, отличным от **subtype**, тогда из A_1**(**X**)** ⊏: A_2**(**Y**)** следует $A_1(X) <: A_2(Y)$.

На основе описанного нами частичного порядка определим три вида отношений между типами.

1. Совместимость
2. Согласованность
3. Тождество

Совместимость – отношение между типами правой и левой части оператора присваивания, а также между типами формальных параметров и соответствующих аргументов вызова. Тип t_1 *совместим с типом* t_2, если $t_1 <: t_2$.

Согласованность – отношение между альтернативами условного выражения, позициями большинства бинарных операций. Типы t_1 и t_2 *согласованы*, если они имеют общую *мажоранту*, т.е. ∃ тип t. $t_1 <: t$ & $t_2 <: t$.

Тождество – отношение между типами результатов вызова и соответствующими формальными результатами. Тип t_1 тождественен типу t_2, если $t_1 <: t_2$ и $t_2 <: t_1$.

6. Детальная семантика языка P

Детально рассмотрим конструкции языка P, позиции которых имеют ограничения на типы соответствующих подконструкций.

6.1. Вызов предиката

Исследуем вызов предиката на примере вызова предиката-функции. Для вызова функции и вызова предиката-гиперфункции рассуждения будут аналогичными.

Синтаксис: CALL := ID **(** [CARG^] **:** CRES^ **)**

Аргумент вызова: CARG := EXPR | TERM

Аргументы вызова предиката представлены в виде списка выражений. Типы конструкций в позициях элементов списка должны быть **совместимы** с соответствующими формальными параметрами.

Результат вызова: CRES := ID | TERM ID | **var** ID

Результаты вызова представлены в виде списка переменных (локальных и глобальных). Типы конструкций в позициях результатов вызова предиката должны быть **тождественны** соответствующим типам, указанным при определении предиката.

6.2. Унарные выражения

Синтаксис: UE := O1 EXPR2

Унарная операция: O1 := **+** | **-** | **!** | **~**

Позиция выражения (EXPR2) может быть *полиморфной* т.е. в зависимости от реализации, конструкции в этой позиции могут иметь разные типы. Так, например, конструкция с унарной операцией «**+**» в позиции выражения может содержать конструкции типа **int** или **real**. Типы конструкции в позиции выражения должны быть **совместимы** с типами, представленными в таблице 5.1.

Таблица 6.1

Операция	*Типы позиций*	*Назначение*
+, -	**int, real**	Унарный плюс и минус
!	**bool**	Логическое отрицание
	Set(T)	Поэлементное дополнение
~	**int**	Побитовое дополнение

6.3. Бинарные выражения

Синтаксис: BE := EXPR O2 EXPR

Бинарная операция: O2 := ***** | **/** | **%** | **+** | **-** | **<<** | **>>** | **in** | **<** | **>** | **<=** | **>=** | **=** | **!=** | **&** | **^** | **or** | **xor** | **=>** | **<=>**

Подобно унарным выражениям во многих бинарных выражениях позиции (EXPR) являются полиморфными.

Типы конструкций в позициях выражений должны быть **совместимы** с типами, указанными в таблице 5.2.

Таблица 6.2

№	*Операция*	*Типы позиций*	*Назначение*
1	<, >, <=, >=	**(nat, nat) (int, int) (real, real)**	арифметическое сравнение
2	=, !=	**(nat, nat) (int, int) (real, real) (list, list)**	проверка на равенство и неравенство
3	/	**(nat, nat) (int, int)**	целая часть от деления
4		**(real, real)**	деление
5	%	**(nat, nat) (int, int)**	остаток от целочисленного деления
6	+	**(nat,nat) (int, int) (real, real)**	арифметическое сложение
7		**(list(T), list(T))**	конкатенация списков
8		**(list(T), T) (T, list(T))**	
9		**(Set(T), Set(T))**	объединение множеств
10		**(array(T, T_1, T_2…) array(T, T_1, T_2…))**	объединение массивов с непересекающимися типами индексов
11	-	**(int, int) (real, real)**	арифметическое вычитание
12		**(Set(T), Set(T))**	разность множеств
13	*	**(nat, nat) (int, int) (real, real)**	арифметическое умножение
14	=>	**(bool, bool)**	импликация
15	<=>	**(bool, bool)**	логическое тождество
16	&	**(Set(T), Set(T))**	пересечение
17		**(nat, nat) (int, int)**	побитовое и
18		**(bool, bool)**	конъюнкция
19	xor	**(Set(T), Set(T))**	симметрическая разность
20		**(nat, nat) (int, int)**	побитовое исключительное или
21		**(bool, bool)**	исключительное или
22	or	**(Set(T), Set(T))**	объединение
23		**(nat, nat) (int, int)**	побитовое или
24		**(bool, bool)**	дизъюнкция
25	^	**(nat, int) (int, int) (real, int)**	возведение в степень
26	in	**(T , Set(T))**	проверка вхождения элемента в множество
27	<<, >>	**(int, int), (nat, nat)**	побитовый сдвиг влево и вправо

В операциях 1-7, 9-25, 27 типы подконструкций в соседних позициях должны быть **согласованы**. Бинарное выражение (операции 1, 2, 26) имеет тип **bool**, в остальных случаях – тип первой позиции.

6.4. Условное выражение

Синтаксис: IFE ::= EXPR ? EXPR : EXPR

Конструкция, стоящая в первой позиции должна быть **тождественна** типу **bool**. Две последние позиции полиморфны по всем типам. Типы конструкций этих позиций должны быть **согласованы**. Условное выражение является единственным тернарным выражением в языке P.

6.5. Условный оператор

Синтаксис: IFOP := **if (**EXPR**)** OP [MOVE] **else** OP [MOVE]

где MOVE – оператор перехода, OP – оператор.

В условном операторе конструкция, стоящая в позиции условия, должна быть **тождественна** типу **bool**.

6.6. Оператор присваивания

Синтаксис: ASSIGN ::= R **=** L

В операторе присваивания тип конструкции правой части должен быть **совместим** с типом левой.

6.7. Инициализация переменной

Синтаксис: INI ::= TERM (ID **=** EXPR)^

При инициализации тип конструкции выражения должен быть **совместим** с типом конструкции переменной.

6.8. Оператор присваивания для struct

Синтаксис: ASSIGN ::= R **= (**(EXPR)^**)**

Переменная в правой части имеет тип **struct**. Типы конструкций выражений списка в левой части должны быть совместимы с соответствующими типами полей в типовом терме **struct**.

Несмотря на то, что типы **struct** с разными порядками полей эквивалентны, в данной конструкции порядок, заданный в описании типа **struct**, важен. Полям структуры присваиваются соответствующее порядку значения из списка выражений.

6.9. Оператор выбора

Синтаксис: **switch (**EXPR**) {**

case EXPR^ : OP [MOVE]

[**default :** OP [MOVE]]

}

Позиции выражений в операторе выбора являются полиморфными. Типы соответствующих конструкций должны быть **совместимы** с одними из следующих типов: (**nat**, **nat,...**) (**int**, **int,...**) (**real**, **real,...**).

Имеется вариант конструкции **switch** для объединений:

switch (EXPR**) {**

case ID [([TP] ID)^]: OP [MOVE]

[**default :** OP [MOVE]]

}

Тип переменной: TP ::= TERM | **var**

В этом случае значением альтернативы будет являться конструктор. Конструкция в позиции выражения должна быть типа **union**.

6.10. Типовые термы

Тип по имени: [ID.] ID [**(**CARG^**)**]

Правила подстановки фактических параметров на место формальных в списке аргументов те же самые, что и для вызова предиката.

Подтип: **subtype (**TERM ID: EXPR | **var** ID: EXPR**)**

Конструкции, стоящие в позиции выражений, должны быть тождественны типу **bool**.

Диапазон: EXPR**..**EXPR

Позиции выражений являются полиморфными. Типы соответствующих конструкций должны быть **совместимы** с одними из следующих типов: (**int**, **int**), (**enum**, **enum**), (**char**, **char**). Первая и вторая конструкции должны быть **согласованы** между собой

6.11. Импорт модуля

Синтаксис: INM::= **import** ID [**(**CARG^**)**] [**as** ID]

Правила подстановки фактических параметров на место формальных те же, что и для вызова предиката.

6.12. Вызов формулы

Синтаксис: CALLF ::= ID (EXPR^)

Типы конструкций в позициях элементов списка выражений должны быть **совместимы** с соответствующими формальными параметрами.

6.13. Вызов процесса

Синтаксис: CALLPROC ::=IPROC **(** [CARG^] (:[CRES^] [MOVE])* **)** ,

где IPROC – имя процесса

При вызове процесса ограничения на типы конструкций аргументов и результатов те же, что и для предикатов.

6.14. Отправка сообщения

Синтаксис: SENDMES ::= **send** IM [(EXPR^)]

где IM – имя сообщения

Типы конструкций в позициях элементов списка выражений должны быть **совместимы** с соответствующими формальными параметрами.

6.15. Литералы

Литералы, представляют собой фиксированное значение, определенного типа данных. Конструкции, являющиеся литералами, имеют следующий синтаксис:

Таблица 6.3

Конструкция	*Тип*
SIGN (DIG)+	**subtype (int** x: x = X**)**
SIGN **0x**(XDIG)+	**subtype (int** x: x = X**)**
[SIGN] (DIG)+ [**.** (DIG)+] [DEG]	**subtype (real** x: x = X**)**
[SIGN] **inf**	**real**
nan	**real**
'SYMB'	**char**
" (SYMB)* **"**	**string**
true | **false**	**bool**
nil	**string**

DIG ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

XDIG ::= DIG | **A** | **B** | **C** | **D** | **E** | **F**

SIGN ::= **+** | **-**

DEG ::= **e** [SIGN] (DIG)+ | **E** [SIGN] (DIG)+

SYMB ::= … | **\"** | **\'** | **** | **\0** | **\n** | **\r** | **\t** | **\x** XDIG [XDIG [XDIG [XDIG]]]

REG ::= (EXPR)^

6.16. Агрегаты

Агрегат определяет конструктор составного объекта. Конструкции, являющиеся агрегатами, имеют следующий синтаксис:

Таблица 6.4

Конструкция	*Тип*
[TERM] **[** REG **]**	**array** (1..n , T) (n равно количеству выражений в REG)
[TERM] **{** REG **}**	**set** (T)
[TERM] **[[** REG**]]**	**list** (T)

REG ::= (EXPR)^

7. Обобщенные типы

Обобщенными типами будем называть:

- Параметры-типы [**type** T]
- Типы **subtype**, в выражение p(x, n) которых явно или неявно будут входить параметры-переменные n [**subtype** (T x: p(x, n))]

В процессе семантического анализа для обобщенного типа накапливаются ограничения на ассоциированный с ним тип. Ограничения будут представлены как совокупность верхних и нижних граней в терминах рассмотренного выше предпорядка типов. Символами "\", "/" будем определять соответственно ограничение сверху и снизу. Далее, **MAX** и **MIN** будут обозначать, соответственно, супер-тип (∀T. T <: **MAX**) и пустой тип (∀T. **MIN** <: T). Каждое ограничение будем заключать в круглые скобки. Если требуется одновременное выполнение ограничений (A) и (B) будем писать (A) ∧ (B). Если требуется выполнение одного из ограничений (A) или (B) будем писать (A) ∨ (B). При определении типа по имени без инициализации ему автоматически будет заданы следующие ограничения {(**MAX**) ∧ (/**MIN**)}. При определении типа **subtype** (T x: p(x,n)) – ограничения {(/**MIN**) ∧ (\T)}. При каждом использовании экземпляров данного типа его ограничения будет изменяться. Сами ограничения также могут содержать обобщенные типы. Ограничения в этом случае будут наследоваться, и при изменении ограничений обобщенного типа будут меняться все ограничения, в которых он используется.

Совокупность ограничений для типа будем называть *концептом.* Концепты позволяют выявлять ошибки в ходе статического анализа типа до инициализации некоторых переменных и типов-переменных. Например, тип-параметр будет некорректным при

следующем ограничении {(/**real**) ∧ (**nat**)}, а тип с ограничением {(**MIN**)} будет пустым, о чем имеет смысл предупредить разработчика. Концепты упрощают использование отдельных модулей программы. Разработчику возможно предоставить формальные ограничения на типы, которые ему разрешено использовать в параметрах модуля. Анализ корректности фактических типов впоследствии будет сведен к проверке соответствующих ограничений на обобщенные типы.

Обобщенные типы обеспечивают языку P поддержку обобщённого программирования [9], которое, не ограничиваясь определенными типами данных, позволяет повторно использовать код, реализующий некоторый алгоритм.

Пусть T – некоторый обобщенный тип с текущим концептом {(P)}, рассмотрим ограничения на тип T для введенных нами ранее отношений на типах.

Таблица 7.1

Отношение на типах	***Концепт***
Тип A совместим с T	{(P) ∧ (/A)}
T совместим с типом A	{(P) ∧ (\A)}
T согласован с типом A с общей мажорантой B	{(P) ∧ (\B)}
T тождественен типу A	{(P) ∧ (\A) ∧ (/A)}

Если последний концепт не противоречив он будет допускать единственный тип A.

В определении концепта чаще всего используются дизъюнкции, но в случаях полиморфных позиций могут использоваться и конъюнкции. Например, для полиморфного оператора «минус» тип операнда будет иметь следующее ограничение: {(**int**) ∨ (**real**) ∨ (**Set**(MAX))}, которое, в свою очередь, эквивалентно ограничению: {(**real**) ∨ (**Set**(MAX))}.

При изменении ограничений для типа следует проверить их непротиворечивость. В этих целях предполагается использование SMT-решателя [6]. Для корректного построения концептов исключается использование ограничений, связанных с рекурсивными типами, следовательно, обобщенные типы не могут быть ассоциированы с рекурсивными типами.

8. Неявная типизация

Язык P имеет элементы неявной типизации, которая, в частности, характеризуется ключевым словом **var**. Существует четыре вида конструкций с неявной типизацией. Для каждой такой конструкции следует восстановить соответствующие типы.

1. **Конструкциях определения массива**

   Определение массива: DM ::= **for (**([TP] ID)^**)** EXPR

Ключевое слово **var** в TP заменяется типовым термом, который соответствует типу индексов конструкции DM типа массив.

Пример:

```
array (int, 1..5) a;
a = for (var i) i*i;
```

2. **Конструкция локальных переменных-результатов**

   CRES := ID | TERM ID | **var** ID

   Типовой терм в данном случае определяется соответствующим типом, указанным при определении вызываемого объекта.

3. **Конструкции case для union**

   switch (EXPR**)**
   {case ID [([TP] ID)^] : OP [MOVE]
   [**default :** OP [MOVE]
   }

   Типы параметров восстанавливаются по типу полей соответствующих конструкторов в типе **union** (EXPR в **switch)**.

4. **Конструкция описания переменных**

   Тип переменной определяется из контекста дальнейшего использования переменной. Для такой переменной в этом случае будет создан новый обобщенный тип, который при использовании этой переменной будет строить свой концепт. По выходу из блока, где использовалась эта переменная, **var** примет значение одной из верхних граней, описанных концептом.

В первых трёх случаях тип восстанавливается непосредственно из контекста конструкции. Для восстановления неявного типа в конструкции 4, используется инструмент обобщенных типов. При обнаружении соответствующей переменной ей присваивается новый обобщенный тип, который по мере использования переменной строит концепт. По выходу из соответствующего блока переменной присваивается тип, который является верхней гранью концепта. Заметим, что концепт должен быть непротиворечив и иметь единственную верхнюю грань, которая отлична от **MIN** и **MAX**.

Описатель **var** в некоторых конструкциях может отсутствовать, в любом случае тип следует восстановить.

9. Задача анализа типов

В процессе трансляции для некоторой синтаксически корректной конструкции требуется установить, является ли эта конструкция семантически правильной. Для этого необходимо провести *семантический анализ*, в частности определить типы для всех подконструкций, выполнить проверку, допустимы ли эти конструкции в данных позициях, определить тип данной конструкции.

Семантический анализ языка P имеет четыре стадии:

1. Выявление неявной типизации и восстановление типов.
2. Определение типов для идентификаторов, литералов и агрегатов.
3. Проверка корректности рекурсивных типов
4. Иерархическая проверка корректности всех подконструкций на соответствующие типовые ограничения позиций.

Вторая стадия, называемая идентификацией, осуществляется при помощи таблиц идентификаторов. Для каждого нового блока или определения предиката создается новая таблица, куда заносятся данные о встречающихся определений переменных и типов.

Для анализа корректности рекурсии параллельно идентификации строиться ориентированный граф. Вершины графа помечены именами типов, дуги типовыми термами. Существование дуги (AB) в этом графе эквивалентно тому, что A непосредственно зависит от B. Если из вершины A существует путь к A, то тип A – рекурсивный. Для рекурсивного типа необходимо найти рекурсивное кольцо и проверить его корректность.

Последняя стадия – проверка типов. Алгоритм проверки типов следующий:

1. Если конструкция имеет подконструкции, то запускаем проверку типов для этих подконструкций.
2. Если конструкция имеет ограничения на тип, то проверяем эти ограничения
3. Для конструкции определяем её тип с использованием типов подконструкций.

Все конструкции, имеющие ограничения на тип, описаны выше. Проверка ограничений возможна, так как все подконструкции будут проверены на корректность и для них будет установлен тип.

10. Заключение

В настоящей работе представлена модель типов языка предикатного программирования, которая включает в себя:

1. Определения рекурсивных типов на базе аппарата наименьшей неподвижной точки рекурсивных типовых уравнений;
2. Предпорядок на типах языка P;
3. Обобщенные типы, построенные на базе концептов – наборов ограничений для типов;
4. Конструкции с неявной типизацией и правила восстановления неявных типов.

Определены три вида отношений: согласованность, совместимость и тождество для пары типов. С помощью данных отношений была формализована семантика конструкций языка предикатного программирования и разработаны правила определения типов выражений и агрегатов. В соответствии с этими правилами разработаны алгоритмы проверки корректности рекурсии, установки типов для языковых конструкций, проверки семантической корректности конструкций [1].

В настоящее время ведется работа по завершению программной реализации анализа типов в экспериментальной системе предикатного программирования.

Список литературы

1. Зубарев А. Ю. Анализ типов в трансляторе с языка предикатного программирования // Материалы 54-й международной научной студенческой конференции МНСК-2016 Математика. Новосибирск: Новосиб. гос. ун-т., 2016. С. 201.
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Версия 0.12. Новосибирск: 2013. 52 с. [Электронный ресурс]. Систем. требования: Adobe Acrobat Reader. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения: 7.12.2016).
3. Клини С. К. Введение в метаматематику: Пер. с англ. М.: 1957. 526 с.
4. Пеленицын А.М. Ассоциированные типы и распространение ограничений на параметры-типы для обобщенного программирования на Scala // Программирование. 2015. №4. С. 13-22.
5. Шелехов В.И. Предикатное программирование: Учеб. пособие. Новосибирск: Новосиб. гос. ун-т, 2009. 109 с.
6. Barrett C., Fontaine P., Tinelli C. The SMT-LIB Standard Version 2.5. 2015. 94 p.
7. Dershem H.L., Jipping M.J. Programming Languages: Structures and Models. Boston: Wadsworth, 1990. 413 p.
8. ISO/IEC 14977:1996(E) Информационная технология – Синтаксический метаязык – Расширенная Форма Бэкуса-Наура (Extended BNF).

9. Musser D.A., Stepanov A.A. Generic Programming // Proceeding of International Symposium on Symbolic and Algebraic Computation. V. 358 of Lecture Notes in Computer Science. Rome. Italy. 1988. P. 1325.
10. Pierce B.C. Types and Programming Languages . Massachusetts: The MIT Press Cambridge, 2002. 623 p.
11. Watt D.A. Programming language concepts and paradigms . Upper Saddle River, New Jersey: Prentice Hall, 1990. 322 p.