Оглавление

Глава 1

Анализ типов

1.1 Что будет, если в нашу систему типов ввести тип $Bool$?

Продублируем изначальные правила:

I	$[[I]] = int$
$E_1 == E_2$	$[[E_1]] == [[E_2]] \wedge [[E_1\ op\ E_2]] = int$
$E_1 op E_2$	$[[E_1]] == [[E_2]] == [[E_1\ op\ E_2]] = int$
$input$	$[[input]] = int$
$X = E$	$[[X]] = [[E]]$
$output\ E$	$[[E]] = int$
$if(E)\ \{S_1\}$	$[[E]] = int$
$if(E)\ \{S_1\}\ else\ \{S_2\}$	$[[E]] = int$
$while\ (E)\ \{S\}$	$[[E]] = int$
$f(X_1, ..., X_n)\ \{...return\ E;\}$	$[[f]] = ([[X_1]], ..., [[X_n]]) \to [[E]]$
$(E)\ (E_1, ..., E_n)$	$[[E]] = ([[E_1]], ..., [[E_n]]) \to [[(E)(E_1, ..., E_n)]]$
$\&E$	$[[\&E]] = \&[[E]]$
$alloc$	$[[alloc]] = \&\alpha$
$null$	$[[null]] = \&\alpha$
$^*E$	$[[E]] = \&[[^*E]]$
$^*X = E$	$[[X]] = \&[[E]]$

Тогда, перво-наперво введём булевый литерал в пару к I - целочисленном литералу:

$$B \Rightarrow [[B]] = boolean$$

Понятно, что возможные значения - это *True* или *False*. Следовательно меняется тип выражений в инструкциях, а также у бинарных операторов - теперь стоило бы выделить логические операторы и арифметические операторы, но т.к. в TIP есть только два логических оператора то нет нужды выписывать какой-нибудь *logOp*.

Ещё одно следствие - мы не знаем тип *input* и *output*. Выпишем изменившиеся правила:

I	$[[I]] = int$
B	$[[B]]] = boolean$
$E_1 \ > \ E_2$	$[[E_1]] == [[E_2]] = int \wedge [[E_1 \ > \ E_2]] = boolean$
$E_1 \ == \ E_2$	$[[E_1]] == [[E_2]] == [[E_1 \ == \ E_2]] = boolean$
$E_1 \ op \ E_2$	$[[E_1]] == [[E_2]] == [[E_1 \ op \ E_2]] = int$
$input$	$[[input]] = \alpha$
$X = E$	$[[X]] = [[E]]$
$output \ E$	$[[E]] = \alpha$
$if(E) \ \{S_1\}$	$[[E]] = boolean$
$if(E) \ \{S_1\} \ else \ \{S_2\}$	$[[E]] = boolean$
$while \ (E) \ \{S\}$	$[[E]] = boolean$
$f(X_1, ..., X_n) \ \{...return \ E;\}$	$[[f]] = ([[X_1]], ..., [[X_n]]) \rightarrow [[E]]$
$(E) \ (E_1, ..., E_n)$	$[[E]] = ([[E_1]], ..., [[E_n]]) \rightarrow [[(E)(E_1, ..., E_n)]]$
$\&E$	$[[\&E]] = \&[[E]]$
$alloc$	$[[alloc]] = \&\alpha$
$null$	$[[null]] = \&\alpha$
$^*E$	$[[E]] = \&[[^*E]]$
$^*X = E$	$[[X]] = \&[[E]]$

1.1.1 Будет ли анализ более полным?

Учитывая, что теперь в инструкциях *if* и *while* условие может быть только типа *Bool*, следовало бы что полнота анализа увеличилась, например можно было бы найти ошибки когда в этих инструкциях условие - это арифметическое выражение, однако семантика языка не совпадает с этим правилом (в обоих инструкциях можно вставить целочисленное значение как условие), поэтому полнота всё-таки упадёт.

1.1.2 Будет ли анализ более точным?

Точность не изменится. Soundness как была такая и осталась.

> *"...if typable, then no runtime type errors occurs..."*

1.2 Что будет, если в нашу систему типов ввести тип $Array$?

По аналогии введём литера массива, а также пустой массив:

$$\begin{gathered} \{\ \} \Rightarrow [[\{\ \}]] = [\alpha] \\ \{E_1, ..., E_n\} \Rightarrow\ [[E_1]] ==\ ...\ == [[E2]] \\ E = \{E_1, ..., E_n\} \Rightarrow\ [[E]] == [\ [[E_1]]\ ] \end{gathered}$$

Все элементы массива долнжы иметь один тип, а вообще-то то есть это либо *int* либо *unit* (пустой массив), либо также массив, обозначим тип массива как $[< typename >]$.

1.2.1 Придумайте правила вывода для новых операторов

Введём операцию взятия индекса:

$$\begin{gathered} E[E_1] \Rightarrow [[E]] == [\alpha]\ \wedge\ [[E_1]] == int\ \wedge\ [[\ E[E_1]\ ]] = \alpha \\ E[E_1] = E_2 \Rightarrow [[E]] == [\alpha]\ \wedge\ [[E_1]] == int\ \wedge\ [[\ E[E_1]\ ]] == [[E_2]]\ \wedge\ E_2 = \alpha \end{gathered}$$

Индексация происходит по числу, следовательно тип индекса - это *int*, также тип присваемого значения должен соответсвовать типу элемента массива, говоря иначе типу выражения, которое возвращает операция взятия индекса.

1.2.2 Попробуйте протипизировать программу

Используя добавленые правила протипизируем программму:

```
main() {
    var x,y,z,t;
    x = {2,4,8,16,32,64};      [[x]]=[  [[2]]  ]=[  int  ]

    y = x[x[3]];               [[3]]=int^x=[int]=>[[x[3]]]=int=>[[x[x[3]]]]=int
                                   =>[[y]]=int

    z = {{},x};                [[{}]]=[a]^[[x]]=[int]=>[[z]]=  [[int]]

    t = z[1];                  [[z]]=[[int]]^[[1]]=int=>[[t]]=[int]

    t[2] = y;                  [[y]]=int ^[[2]]=int ^[[t]]=[int]=>[[t[2]]]=int
}
```

В результате получаем:

$$
\begin{gathered}
[[x]] = [\ int\ ] \\
[[y]] = int \\
[[z]] = [\ [\ int\ ]\ ] \\
[[t]] = int
\end{gathered}
$$

1.3 Дополнительные задания

Exercise 3.9, p. 24: This exercise demonstrates the importance of the term equality axiom. First explain what the following TIP code does when it is executed:

```
var x,y;
x = alloc 1;
y = alloc (alloc 2);
x = y;
```

Then generate the type constraints for the code, and apply the unification algorithm (by hand).

Что делает этот код:

1. Объявляет две перемеенные X и Й
2. Аллоцирует ячейку памяти со значением 1 (X - поинтер)

3. Аллоцирует ячейку памяти со значением аллокации ячейки памяти со значением 2, иначе говоря Й - это птр на птр со значением 2

4. Х приравнивается к Й $\rightarrow$ икс теперь хранит ссылку на п.3

Протипизируем программу:

```
var x,y;                    //
x = alloc 1;                // [[x]] = [[alloc 1]]
y = alloc (alloc 2);        // [[y]] = [[alloc [[alloc 2]] ]]
x = y;                      // [[x]] = [[y]]
```

Получаем:

```
[[x]] = &int
[[y]] = &&int
[[x]] /= [[y]]
```

1.4 Результат реализации TypeAnalysis в TIP

Тестировал на программе:

```
g(a){
    return *a;
}

f(){
    var a;
    var b;
    a=10;
    if( a == 10 ){
        b=g(&a);
    }

    return a;
}
```

Вывод анализа:

```
[info] Inferred types:
  ⟦a[6:9]⟧ = int
  ⟦b[7:9]⟧ = int
  ⟦g(a[1:3]){...}:1:1⟧ = (↑int) -> int
  ⟦(*a)[2:12]⟧ = int
  ⟦a[1:3]⟧ = ↑int
  ⟦g(&a)[10:11]⟧ = int
  ⟦f(){...}:5:1⟧ = () -> int
  ⟦&a[10:13]⟧ = ↑int
  ⟦10[8:7]⟧ = int
  ⟦10[9:14]⟧ = int
  ⟦(a == 10)[9:10]⟧ = int
[info] Results of types analysis of examples/locals.tip written to ./out/locals.tip__types.ttip
[success] Total time: 1 s, completed Jun 4, 2025, 11:27:06 PM
```

Рис. 1.1: Результат TypeAnalysis

Глава 2

Теория решёток

2.1 Почему нам не подходит конкретный домен?

Для начала я обратился к [1] чтобы понять вообще разницу между абстрактным и конкретным доменами (пусть и кажется оно интуитивным). Там приводится пример *complete lattice*, которую называюь *concrete semantic domain*. Используется анализ множества возможных состояний программы, имея такую решётку и функции трансформации можно теоритически вывести все 'контексты', возможные состояния программы, однако это было бы просто невычислимо, учитывая точность такого анализа.

Если бы нам хотелось использовать конкретный домен для определения знака в выражении, тогда мы могли бы выбрать например домен целых чисел. Тогда необходимо было бы знать конкретное значение числа для переменной, что конечно делает анализ бесконечно сложным в теории.

2.2 У решетки есть максимальный и минимальный элементы...

2.2.1 Являются ли они точной верхней или нижней гранью какого-либо подмножества S?

Да! Доказать это можно через следующие определения [2]:

> A *semilattice* is a partial order $(X, \leq)$ in which every doubleton $\{x, y\}$ has a least upper bound, denoted $x \vee y$ and called the *join* of x and y. Even though the relation $\leq$ is partial (i.e., not linear as an order), the operation $\vee$ is total ($x \vee y$ is well-defined for all elements $x, y \in X$).

> With this definition there is a dual notion, that of lower semilattice (so “semilattice” in the above means “upper semilattice”), in which every doubleton has a greatest lower bound, denoted x ∧ y and called their *meet*.

Определение решётки затем складывается из нижней полурешётки и верхней полурешётки:

> A *lattice* is a poset that is simultaneously an *upper semilattice* and a *lower semilattice*.

Для *upper semilattice* и *lower semilattice* определены *join* и *meet*, следовательно они присущи и решётке.

Из чего можно сделать вывод: решётка всегда имеет как точную верхнюю грань, так и точную нижнюю грань для любого конечного подмножества S, поскольку операции *join* и *meet* определены для всех пар элементов. Максимальный элемент решётки является точной верхней гранью всего множества, а минимальный — его точной нижней гранью.

2.2.2 Уникальны ли они?

Least upper bound и *greatest lower bound* решётки должны быть уникальны, но просто *upper bound* и *lower bound* для каких-либо {a, b} не обязаны быть уникальными.

2.3 Произведения решёток

2.3.1 Как выглядит $\sqcup$ и $\sqcap$ для $L_1 \times L_2 \times ... \times L_n$?

Напомню сам себе определения:

- $\sqcup L$ - $X \sqsubseteq \bigsqcup X \vee \forall y \in S : X \sqsubseteq y \Rightarrow \bigsqcup X \sqsubseteq y$
- $\sqcap L$ - $\bigsqcap X \sqsubseteq X \vee \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \bigsqcap X$

Достаточно посмотреть на получившуюся после перемножения решётку:

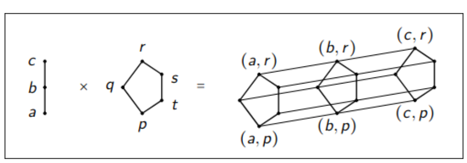

Рис. 2.1: Перемножение решёток

На ней и видно, что операции $\sqcup$ и $\sqcap$ определяются *покоординатно* следующим образом:

$$(a_1, a_2, \dots, a_n) \sqcup (b_1, b_2, \dots, b_n) = (a_1 \sqcup b_1,\ a_2 \sqcup b_2,\ \dots,\ a_n \sqcup b_n)$$

$$(a_1, a_2, \dots, a_n) \sqcap (b_1, b_2, \dots, b_n) = (a_1 \sqcap b_1,\ a_2 \sqcap b_2,\ \dots,\ a_n \sqcap b_n)$$

А $\top$ и $\bot$ - для $productL$ - кортеж из получившихся $\top$ и $\bot$

2.3.2 Какая высота у произведения решеток $L_1 \times L_2 \times ... \times L_n$?

$$height(L_1 \times \dots \times L_n) = height(L_1) + \dots + height(L_n)$$

2.4 Точная верхняя/нижняя грань решётки отображений

Пусть f — функция отображения. Тогда:

$$f_\top(x) = \top \forall x \in A,$$
$$f_\bot(x) = \bot \forall x \in A.$$

Эти отображения являются соответственно наибольшим и наименьшим элементами решётки отображений $A \to L$.

2.5 Высота решётки отображений

Высота решётки отображений $A \to L$ выражается через мощность области определения A и высоту базовой решётки L:

$$height(A \to L) = |A| \cdot height(L).$$

2.6 Почему нельзя использовать унификационный решатель?

Потому что нам интересны уравнения вида:

$$x_n \sqsubseteq f_n(x_1, x_2, ..., x_n)$$

Нам интересно чтобы результат в каждоый ноде был меньше чем соотв. функция в которую мы его передали. Для такого унификационные солверы не подходят, потому что много *ambiguity*

2.7 Результат реализации SignAnalysis в TIP

Тестил над файлом:

```
fun(x) {
    var y;
    var k ;
    k = 8;
    y = 7;
    while(k > y) {
        k = k - 1;
    }
    return 0;
}
main() {
    var pos, neg, top, zero;
    var later;
    pos = 5 + 5;
    pos = 5 * 5;
    neg = -5 - 5;
    neg = -5 * 5;
    neg = 5 * -5;
    top = 5 - 5;
    top = top * 5;
    zero = top * 0;
    zero = 5 * zero;
    later = 7;
    return 0;
}
```

Вывод в виде графа:

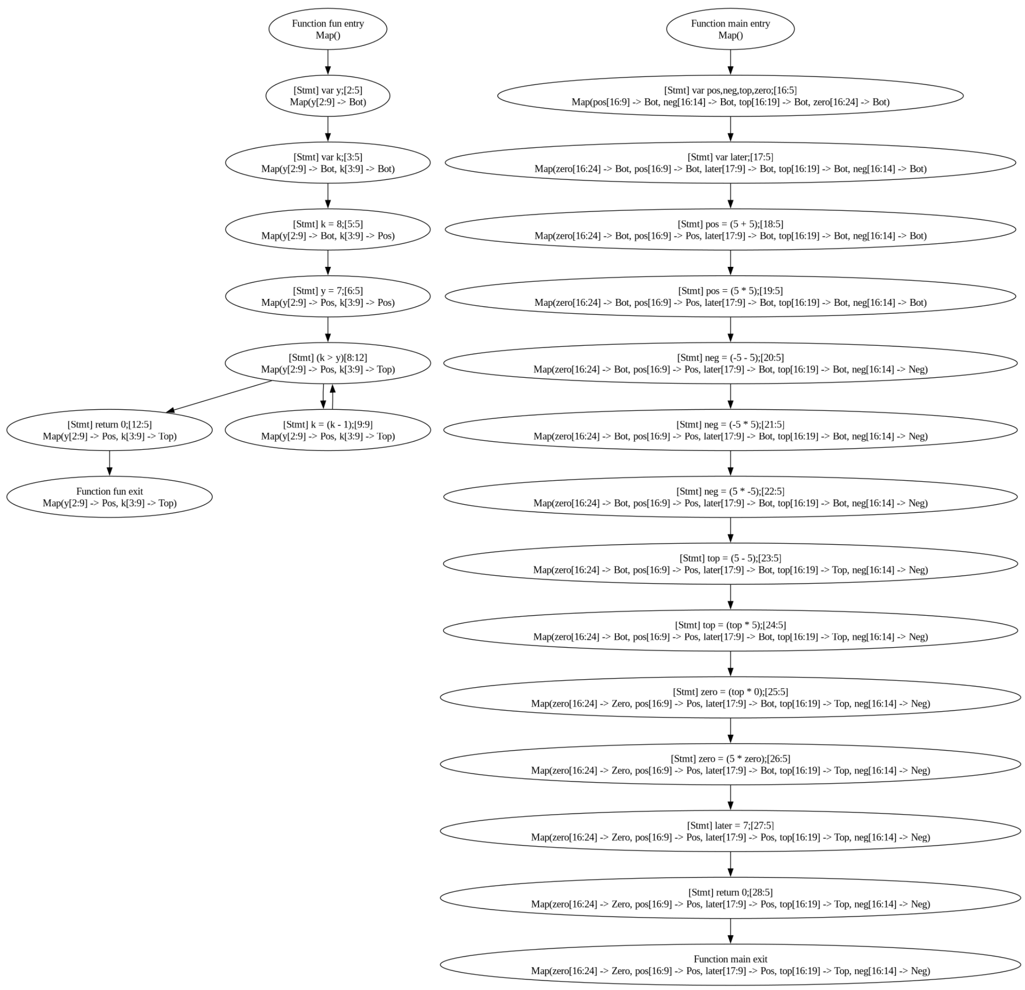

Рис. 2.2: Результат SignAnalysis

Список литературы

[1] Stefan Bydge. «Abstract Interpretation and Abstract Domains». В: *Department of Computer Science and Electronics M¨alardalen University V¨asteras, Sweden* (2006).

[2] *Lattice Theory*. URL: http://boole.stanford.edu/cs353/handouts/book1.pdf.

Глава 3

Потокочувствительные анализы

3.1 Какова сложность структурного алгоритма?

Пусть n — количество узлов CFG. Алгоритм последовательно обходит все узлы графа, и для каждого из них выполняет операции добавления или удаления информации о переменных. Каждая такая операция занимает время $O(k)$, где k — количество переменных в программе (или высота решётки, если говорить точнее).

При изменении состояния узла он может быть снова добавлен в *worklist* для пересчёта зависимых узлов. В худшем случае это приведёт к k проходам по всем n узлам графа — именно столько шагов нужно, чтобы достичь фикспоинта для всех переменных.

Таким образом, временная сложность алгоритма составляет:

$$O(n \times k^2)$$

Если в графе нет циклов (то есть, CFG ацикличен), то каждый узел обрабатывается только один раз. Это исключает повторные проходы по графу, и временная сложность снижается до:

$$O(n \times k)$$

Что касается сложности по памяти: нам нужно хранить состояние (значения переменных) для каждого узла программы. Общая память, таким образом, составляет:

$$n \times k$$

т.к. для каждого из n узлов хранится информация о k переменных.

3.2 Решите систему ограничений

Операции для *very busy analysis*:

$$[[x = E]] = removerefs(JOIN(n), x) \cup exprs(n)$$

$[[n]] = JOIN(n) \cup exprs(n)$ - для любых других n

$$JOIN(v) = \bigcap_{w \in succ(v)} [[w]]$$

Пример:

```
1     var x, a, b;
2     x = input;
3     a = x - 1;
4     b = x - 2;
5     while(x > 0) {
6         output a*b -x;
7         x = x - 1
8     }
9   output a*b
```

Результат:

```
1 {}
3 {x-1}
4 {x-1, x-2}
5 {x-1, x-2, x > 0}
6 {a*b-x, a*b, x-1, x-2, x > 0}
7 {a*b, x-1}
9 {a*b, x-1}
```

3.3 Результат реализации Live Variables Analysis

Тестировал на:

```
main() {
  var a,b,i;
  a = 5;
  b = 42;
  i = a;
  while (b > i) {
    // ...
    i = i + 1;
  }
  return 0;
}
```

Вывод:

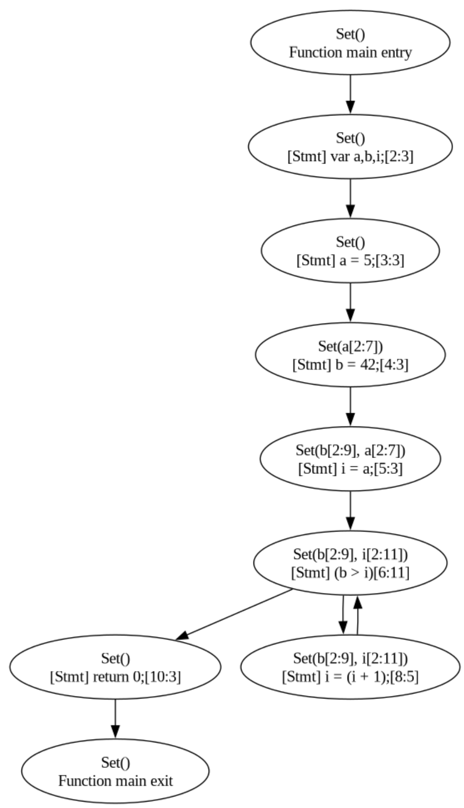

Рис. 3.1: Результат Live Vars Analysis

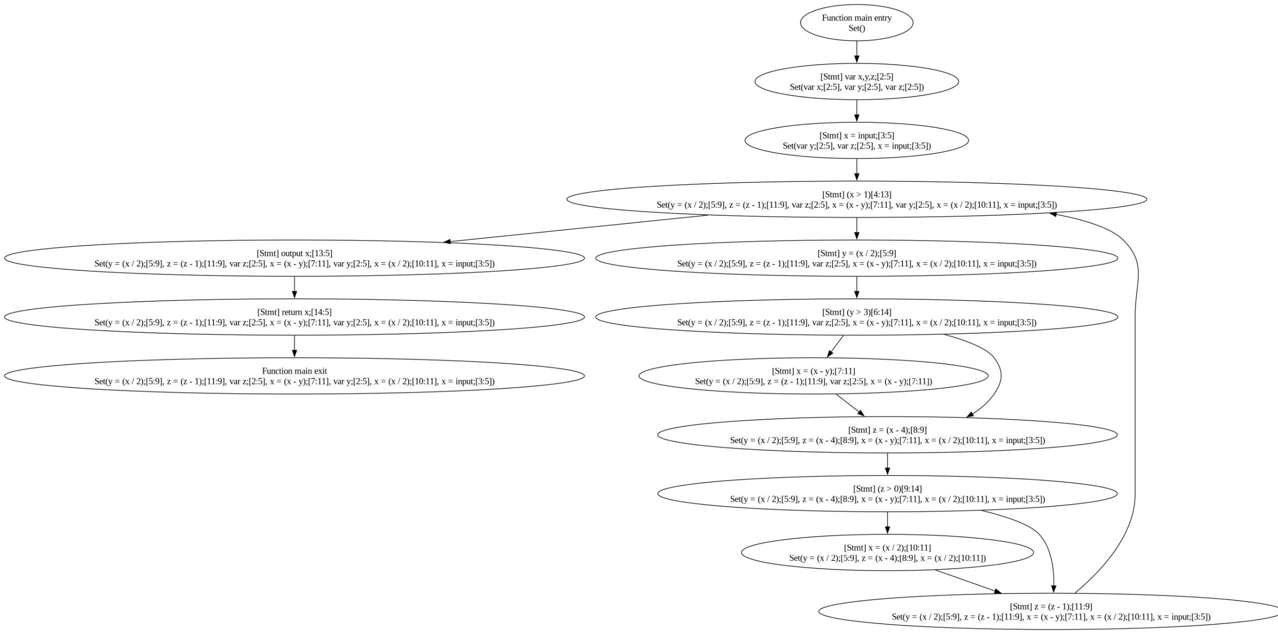

Рис. 3.2: Результат RD Analysis

3.4 Результат реализации Reaching Definitions Analysis

Тестил на программе со слайда:

```
main() {
    var x, y, z;
    x = input;
    while(x > 1) {
        y = x / 2;
        if (y > 3) x = x - y;
        z = x - 4;
        if (z > 0) x = x / 2;
        z = z - 1;
    }
    output x;
}
```

Вывод:

Глава 4

Интервальный анализ

4.1 Интервальная арифметика

Допустим для компилятора требуется информация о размерах переменных:

1. bool (1 bit)
2. byte (8 bit signed)
3. char (16 bit unsigned)
4. int (32 bit signed)
5. bigint (any integer)
6. any (any thing)

$(T)E$ - Операция приведения типов

4.1.1 Какую решётку использовать?

Интервальная решётка с widening, для операции:

$$w([a,b]) = [max\{i \in B | i \leqslant a\}, min\{i \in B | i \leqslant b\}]$$

возьмём множество степеней двойки (где min и max это они же для int):

$$B = \left\{ -2^i \cup\ 2^i \ \middle|\ i = 0, \ldots, log_2(INT\_MAX) \right\}$$

Update!

Я додумался, что можно взять не все степени двойки а только те которые нужные, те:

$$B = \{minsize(Boolean), maxsize(Boolean), minsize(Byte), maxsize(Byte), minsize(Char), ...\}$$

Но ещё надо расширить для опредения типа, поэтому помимо интервала добавим в решётку τ:

$$\text{TNothing} \prec \text{TBool} \prec \text{TByte} \prec \text{TChar} = \text{TInt} \prec \text{TBigInt} \prec \text{TAny}$$

Тип будем определять по интервалу:

$$\text{resolveType}(l, h) = \begin{cases} \text{TBool} & \text{если } 0 \leq l \wedge h \leq 1 \\ \text{TByte} & \text{если } -128 \leq l \wedge h \leq 127 \\ \text{TChar} & \text{если } 0 \leq l \wedge h \leq 65535 \\ \text{TInt} & \text{если } -2^{31} \leq l \wedge h \leq 2^{31} - 1 \\ \text{TBigInt} & \text{иначе} \end{cases}$$

4.2 Результат реализации Variable Size Analysis в TIP

Пример:

```
main() {
    var x,y;
    x = 100;
    y = input;
    while (y > x) {
        x = 7;
        x = x+1;
        y = y+1;
    }
    return 0;
}
```

Вывод:

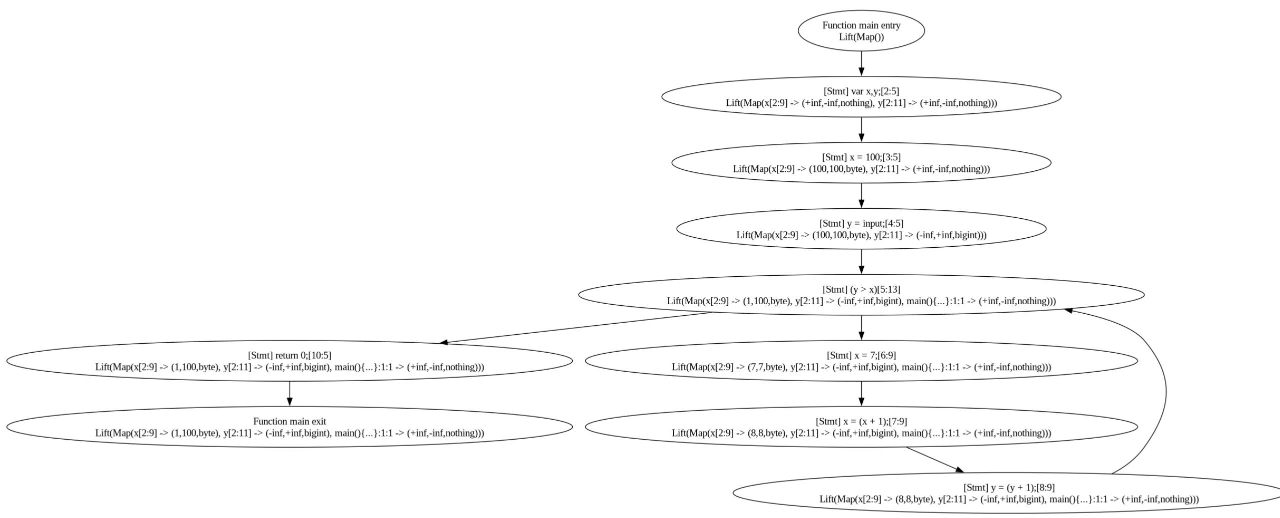

Рис. 4.1: Результат VariableSizeAnalysis