

Оглавление

1	Анализ типов	2
1.1	Что будет, если в нашу систему типов ввести тип <i>Bool</i> ?	2
1.1.1	Будет ли анализ более полным?	4
1.1.2	Будет ли анализ более точным?	4
1.2	Что будет, если в нашу систему типов ввести тип <i>Array</i> ?	4
1.2.1	Придумайте правила вывода для новых операторов	4
1.2.2	Попробуйте протипизировать программу	5
1.3	Дополнительные задания	5
1.4	Результат реализации TypeAnalysis в TIP	6
2	Теория решёток	8
2.1	Почему нам не подходит конкретный домен?	8
2.2	У решетки есть максимальный и минимальный элементы...	8
2.2.1	Являются ли они точной верхней или нижней гранью какого-либо подмножества S?	8
2.2.2	Уникальны ли они?	9
2.3	Произведения решёток	9
2.3.1	Как выглядит \sqcup и \sqcap для $L_1 \times L_2 \times \dots \times L_n$?	9
2.3.2	Какая высота у произведения решеток $L_1 \times L_2 \times \dots \times L_n$?	10
2.4	Точная верхняя/нижняя грань решётки отображений	10
2.5	Высота решётки отображений	10
2.6	Почему нельзя использовать унификационный решатель?	10
2.7	Результат реализации SignAnalysis в TIP	10

Глава 1

Анализ типов

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

Продублируем изначальные правила:

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

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

$$B \Rightarrow [[B]] = \text{boolean}$$

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

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 литерал массива, а также пустой массив:

$$\begin{aligned} \{ \} &\Rightarrow [[\{ \}]] = [\alpha] \\ \{E_1, \dots, E_n\} &\Rightarrow [[E_1]] == \dots == [[E_n]] \\ E = \{E_1, \dots, E_n\} &\Rightarrow [[E]] == [[[E_1]]] \end{aligned}$$

Все элементы массива должны иметь один тип, а вообще-то то есть это либо *int* либо *unit* (пустой массив), либо также массив, обозначим тип массива как $\langle \text{typename} \rangle$.

1.2.1 Придумайте правила вывода для новых операторов

Введём операцию взятия индекса:

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

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

1.2.2 Попробуйте протипизировать программу

Используя добавленные правила протипизируем программу:

```
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:

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

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 переменные X и Y
2. Аллоцирует ячейку памяти со значением 1 (X - поинтер)

3. Аллоцирует ячейку памяти со значением аллокации ячейки памяти со значением 2, иначе говоря \dot{Y} - это птр на птр со значением 2
4. X приравнивается к $\dot{Y} \rightarrow$ 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 TIR

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 называю *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 $x \wedge y$ 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 $L_1 \times L_2 \times \dots \times L_n$?

Напомню сам себе определения:

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

Достаточно посмотреть на получившуюся после перемножения решётку:

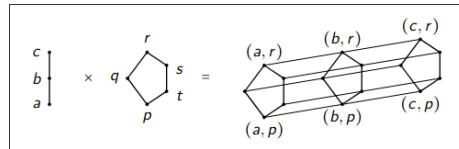


Рис. 2.1: Перемножение решёток

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

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

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

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

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

2.4 Точная верхняя/нижняя грань решётки отображений

Пусть f — функция отображения. Тогда:

$$f_{\top}(x) = \top \forall x \in A,$$

$$f_{\perp}(x) = \perp \forall x \in A.$$

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

2.5 Высота решётки отображений

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

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

2.6 Почему нельзя использовать унификационный решатель?

Потому что у нас *data-flow sensitive* анализ, надо знать о предыдущих состояниях. Особенно когда, например, мы приходим из разных веток *if-else*. Поэтому у нас и существует *JOIN* операция.

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:

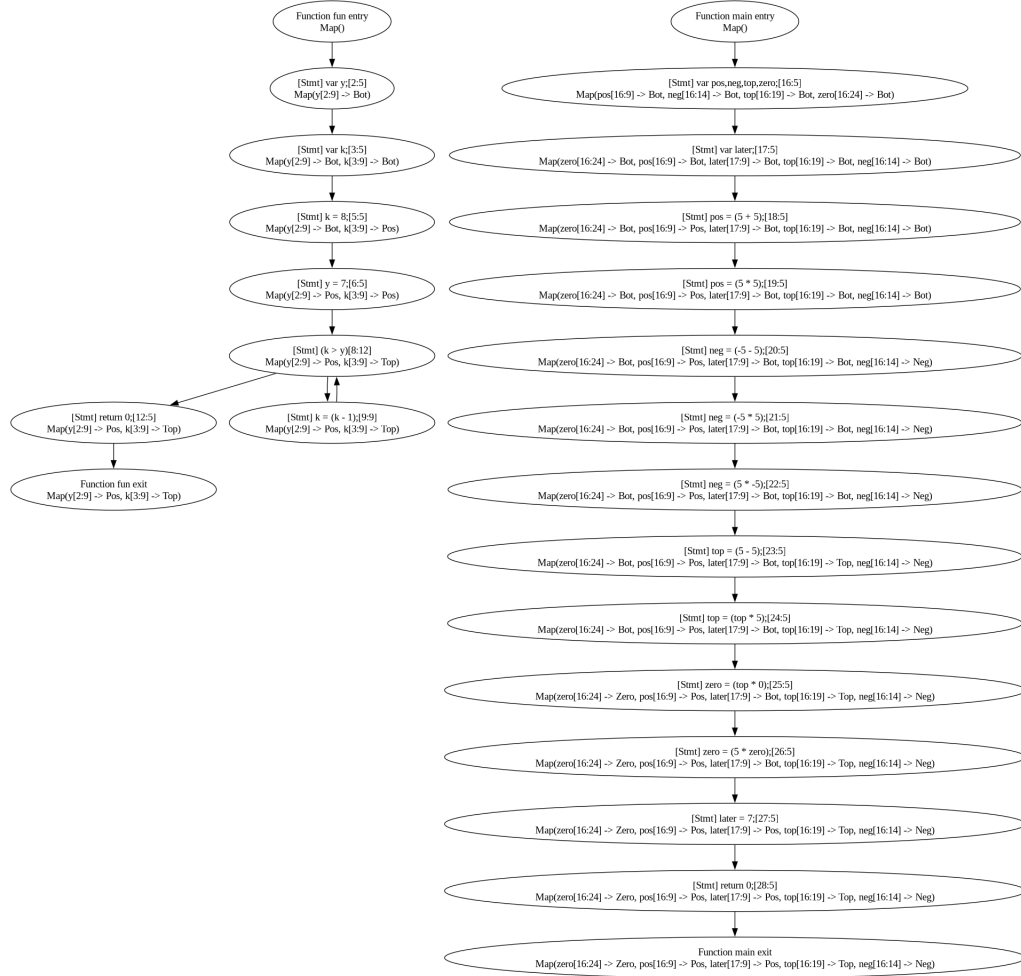


Рис. 2.2: Резултат SignAnalysis

Список литературы

- [1] Stefan Bydger. «Abstract Interpretation and Abstract Domains». B: *Department of Computer Science and Electronics Mälardalen University Västerås, Sweden* (2006).
- [2] *Lattice Theory*. URL: <http://boole.stanford.edu/cs353/handouts/book1.pdf>.