Потоковый вывод в конструкторе компиляторов RiDE

Михаил Бахтерев
m.bakhterev@imm.uran.ru

Институт Математики и Механики им. Н.Н. Красовского

ЯПК, Ростов-на-Дону, 2017

Модель программирования RiDE-3

РАспределёный **И**нтуиционистский **Д**вижок

- Тьюринг-полные открытые потоки данных.
- Ресурсопластичность.
- Выразительность и простота:
  - близость к λ-исчислению;
  - рекурсивность;
  - неопределённость только по необходимости;
  - однородное описание потоков между сочетаниями:

$$\{\text{CPU, GPGPU, MIC, FPGA, host, cluster, web, } \ldots\}^2.$$

- Высокая производительность:
  - совмещённые рассчёты и обмены данными;
  - отсутсвие избыточных копирований;
  - распределённый рассчёт потока управления.
- Независимость от:
  - языка программирования
  - и аппаратуры: FPGA, сетевые GPGPU, APU, ...

Идеальное решение должно существовать

ДЗК структур событий

(E – события, $\leq$ – причинные связи, $\vdash$ – конфликты)

Декартово замкнутые категории

- Автоматически интуицинистская логика.
- Автоматически λ-исчисление.
- Авторы нашли только CSP в особом классе структур.

РАЙД API

Примитивы обмена данными

1. Блок данных.
2. Контакт
   - `(pin :a :b :c "hello")`.
3. Очередь
   - `(zip (str (+ 1 2)) "kitty")`.

Указатели на распределённые структуры

4. Диапазон путей
   - `(! pin :a (span "") :c)`
   - `(? pin (span "x y z" :Z))`
   - `(! zip (str (+ 1 2)) "kitty")`
   - `(? zip)`.

РАЙД API

Поток вычисления

5. Продолжение.
6. Процесс.

Операции

7. Запись блока данных
   - `(:= (! pin :a) "hello kitty")`.
8. Размещение продолжения
   - `(run + 1 (pin :a) (! pin :c))`.
9. Системные процедуры:
   - `fork`, `done`, `update`.

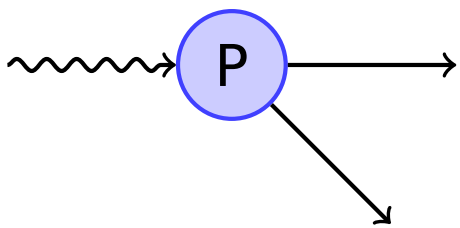
Сборка мусора

Примитивы для обмена данными

- Блоки данных:
  - массивы данных
  - с уникальными идентификаторами.

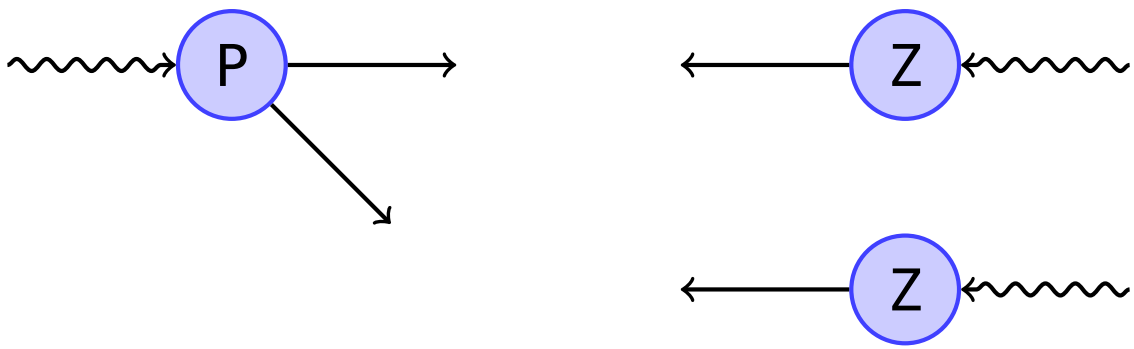
Примитивы для обмена данными

- Блоки данных:
  - массивы данных
  - с уникальными идентификаторами.
- Контакты **`(pin :P)`**:
  - распределённые ssa-переменные;
  - источник определённости.
- Задаются путями:
  - обычные списки строк;
  - откладываются от корней (просто УИД).



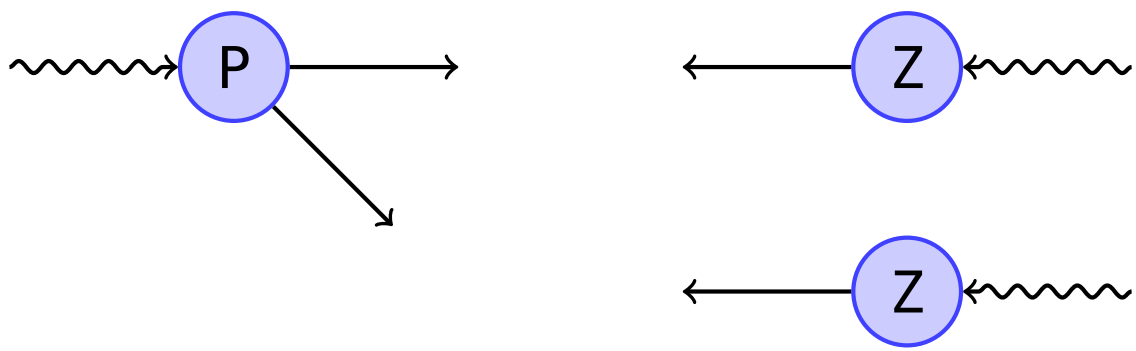
Примитивы для обмена данными

- Блоки данных:
  - массивы данных
  - с уникальными идентификаторами.
- Контакты **`(pin :P)`**:
  - распределённые ssa-переменные;
  - источник определённости.
- Задаются путями:
  - обычные списки строк;
  - откладываются от корней (просто УИД).



Примитивы для обмена данными

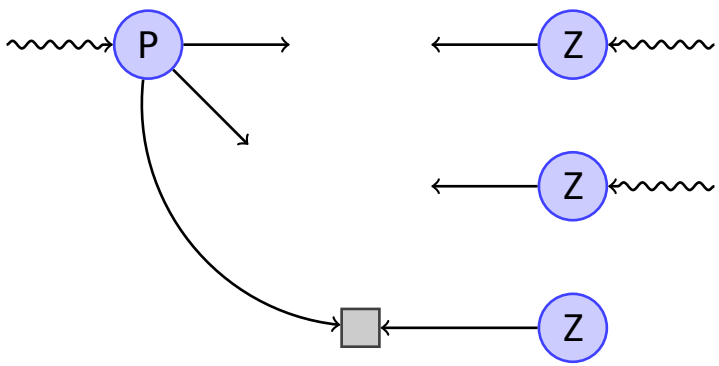
- Очереди `(zip :Z)`:
  - переменчивая идентичность;
  - открытость системы;
  - «застёжка» чтений и записей.
- Задаются путями.



Все вычисления *параллельны, распределены и локальны*.

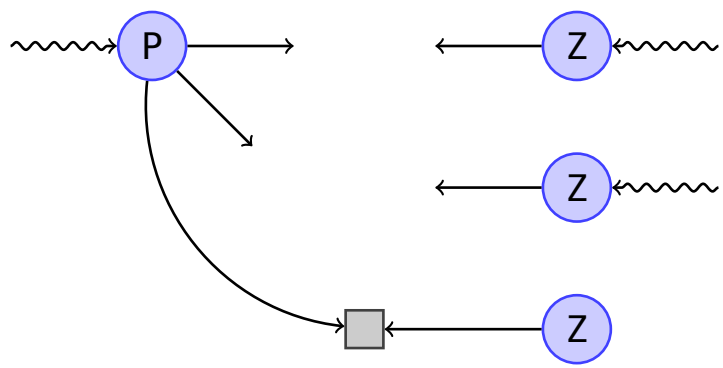
Продолжения

- Обычные продолжения.
- Но в РАЙД они формируются массово и параллельно.
- Активируются асинхронно по готовности аргументов.
- На самом деле, ничего особенного.
- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`



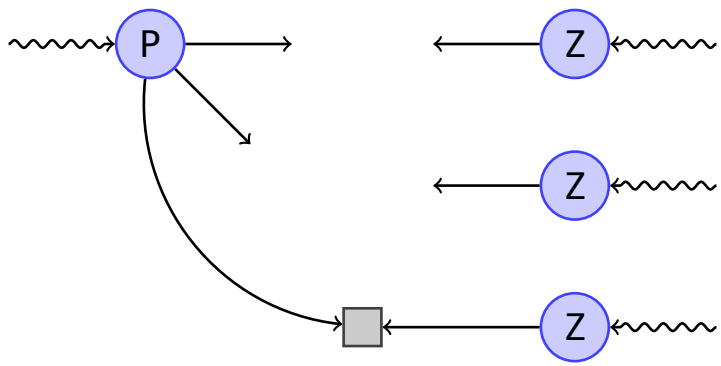
Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`



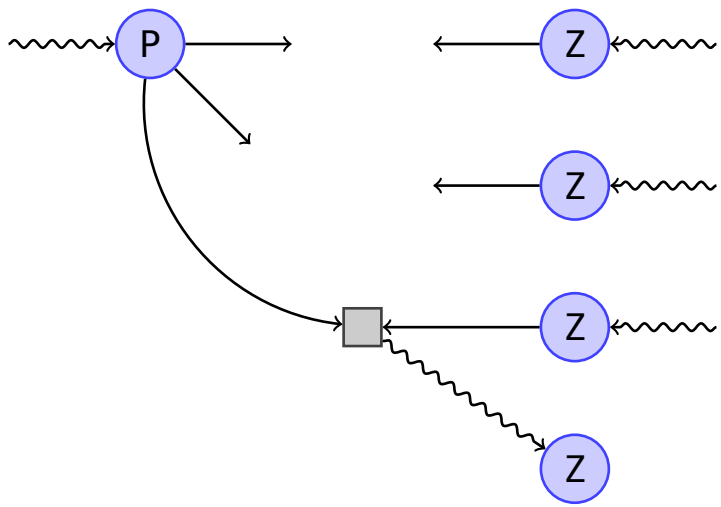
Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`
- Выращивание графа:
  - `(define (F hk p z z-ref)`



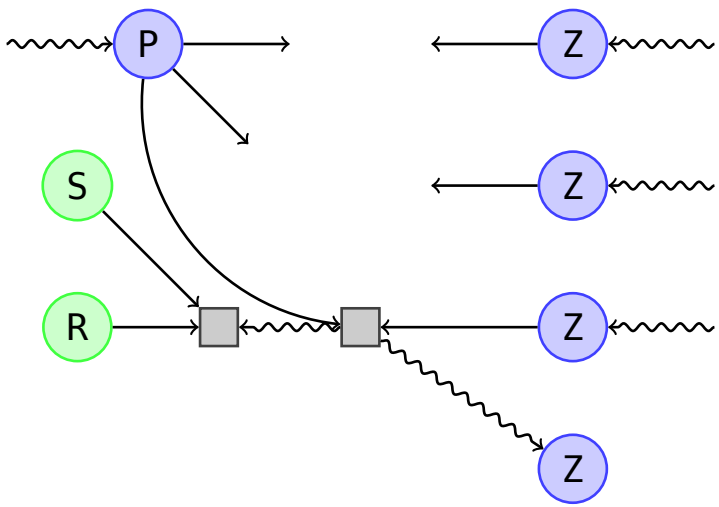
Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`
- Выращивание графа:
  - `(define (F hk p z z-ref)`
    - `(:= z-ref (+ p z))`



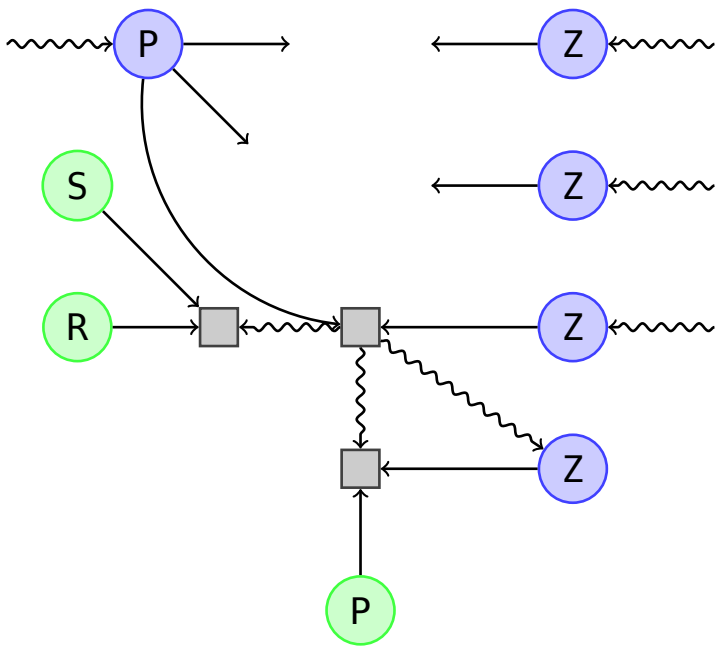
Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`
- Выращивание графа:
  - `(define (F hk p z z-ref)`
    - `(:= z-ref (+ p z))`
    - `(run h (pin :R) (pin :S))`



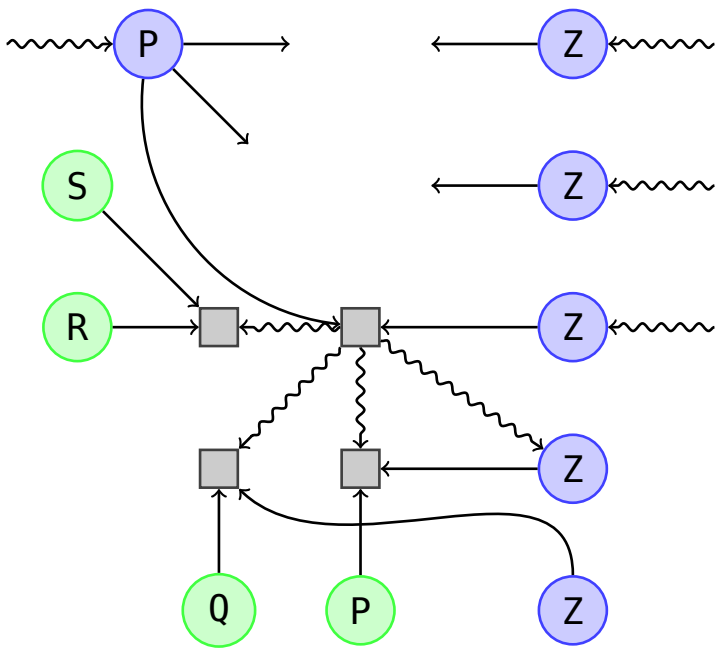
Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`
- Выращивание графа:
  - `(define (F hk p z z-ref)`
    - `(:= z-ref (+ p z))`
    - `(run h (pin :R) (pin :S))`
    - `(run g (zip z-ref) (pin :P))`



Формирование графа через продолжения

- `(run F "hello kitty" (pin :p) (zip :z) (! zip :Z))`
- Выращивание графа:
  - `(define (F hk p z z-ref)`
    - `(:= z-ref (+ p z))`
    - `(run h (pin :R) (pin :S))`
    - `(run g (zip z-ref) (pin :P))`
    - `(run f (zip z-ref) (pin :Q)))`



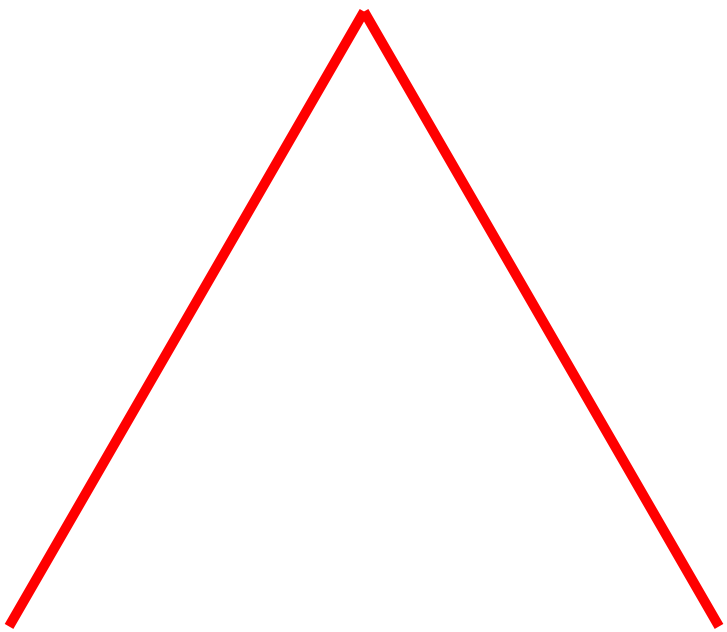
Управление *распределённое*, рассчёты *локальны*

Области: распределённый контроль потока

- Все **`pin`** и **`zip`** независимы, работа с ними параллельна.
- Необходим механизм работы со структурами данных:
  - указание на структуры;
  - передача в процедуры;
  - обновление больших структур;
  - *рекурсия*:
    - кадры активации процедур,
    - произведение для ДЗК.
- Области.
  - 4 варианта: {**`?`**-чтение, **`!`**-запись} $\times$ {**`pin`**, **`zip`**}.
  - Просто блок данных, отвечающий на два вопроса.
    1. Попадает ли путь в диапазон?
    2. Если попадает, от какого он корня идёт?

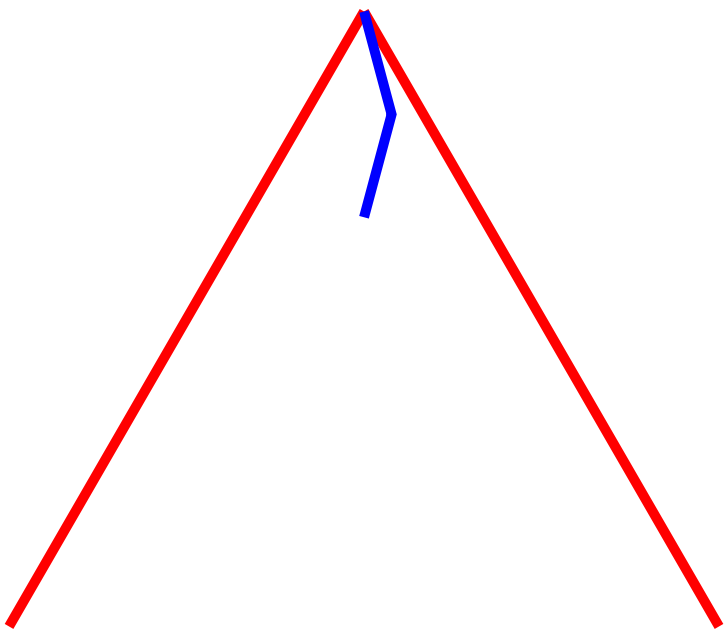
Конструкторы областей

- (! pin A



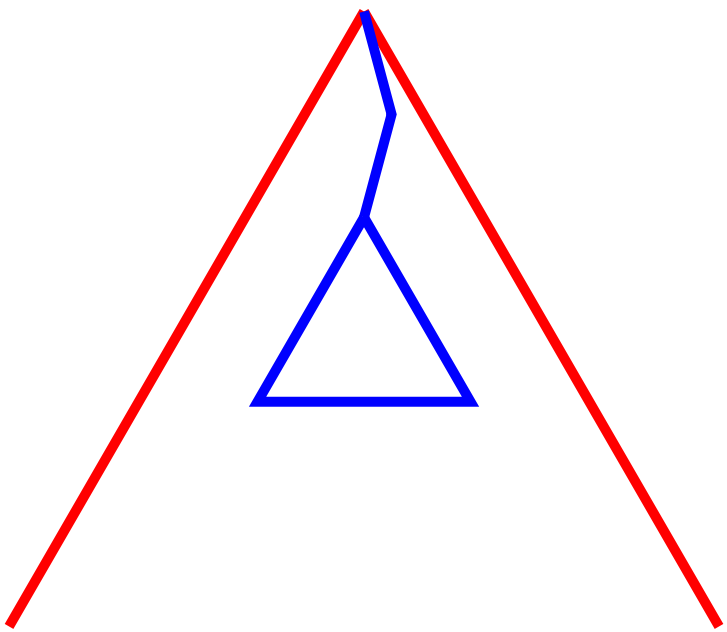
Конструкторы областей

- (! pin A
  - :a :b



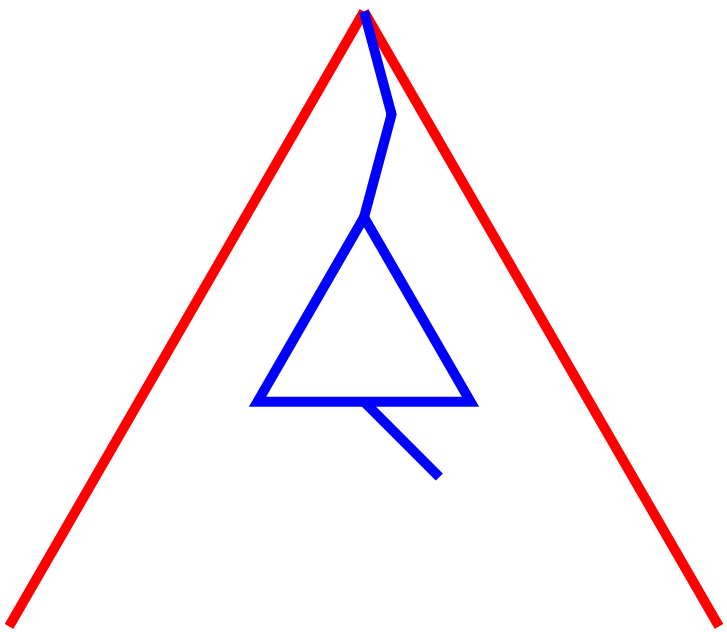
Конструкторы областей

- `(! pin A`
  - `:a :b`
  - `(span :c :d)`



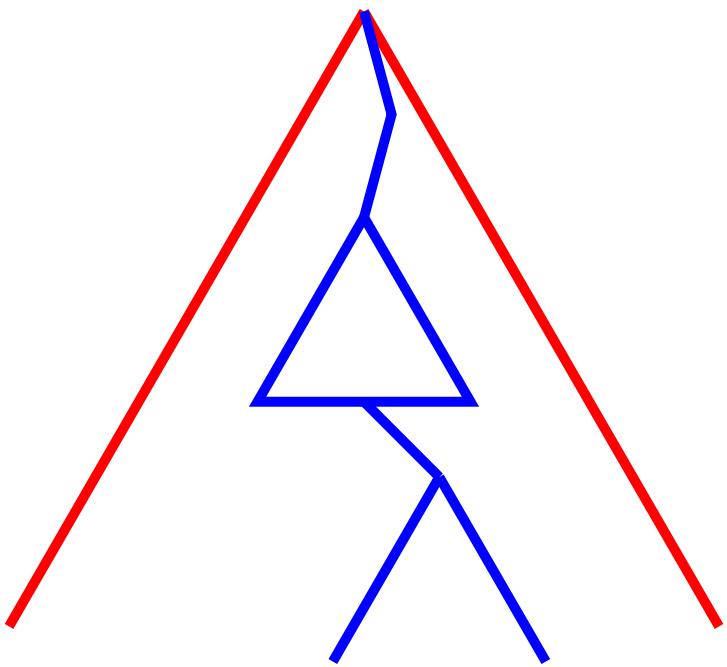
Конструкторы областей

- (! pin A
  - :a :b
  - (span :c :d)
  - :e



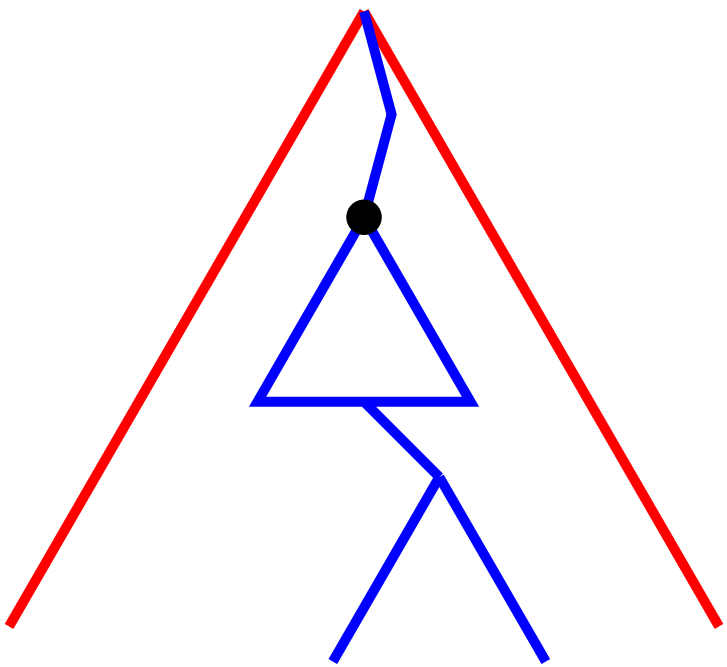
Конструкторы областей

- (! pin A
  - :a :b
  - (span :c :d)
  - :e
  - (span ””))



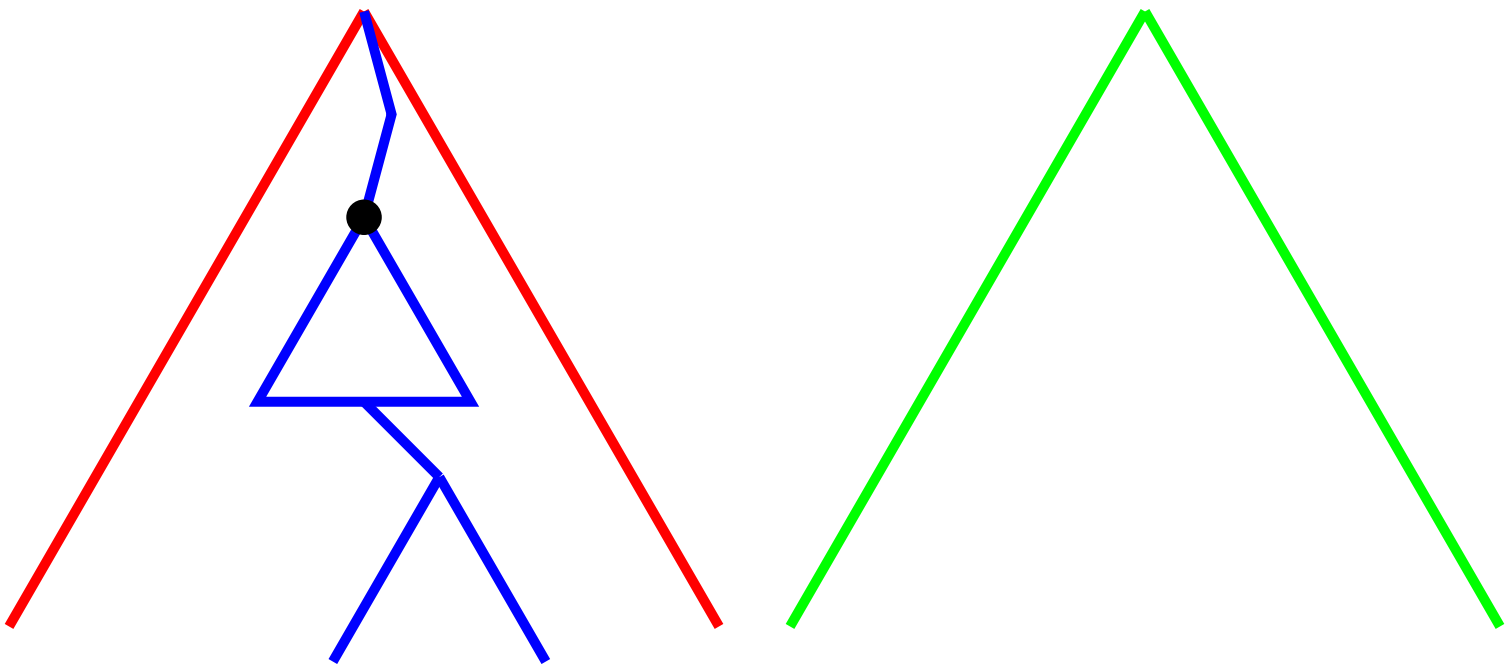
Конструкторы областей

- (! pin A
  - :a :b
  - (span :c :d)
  - :e
  - (span ””))



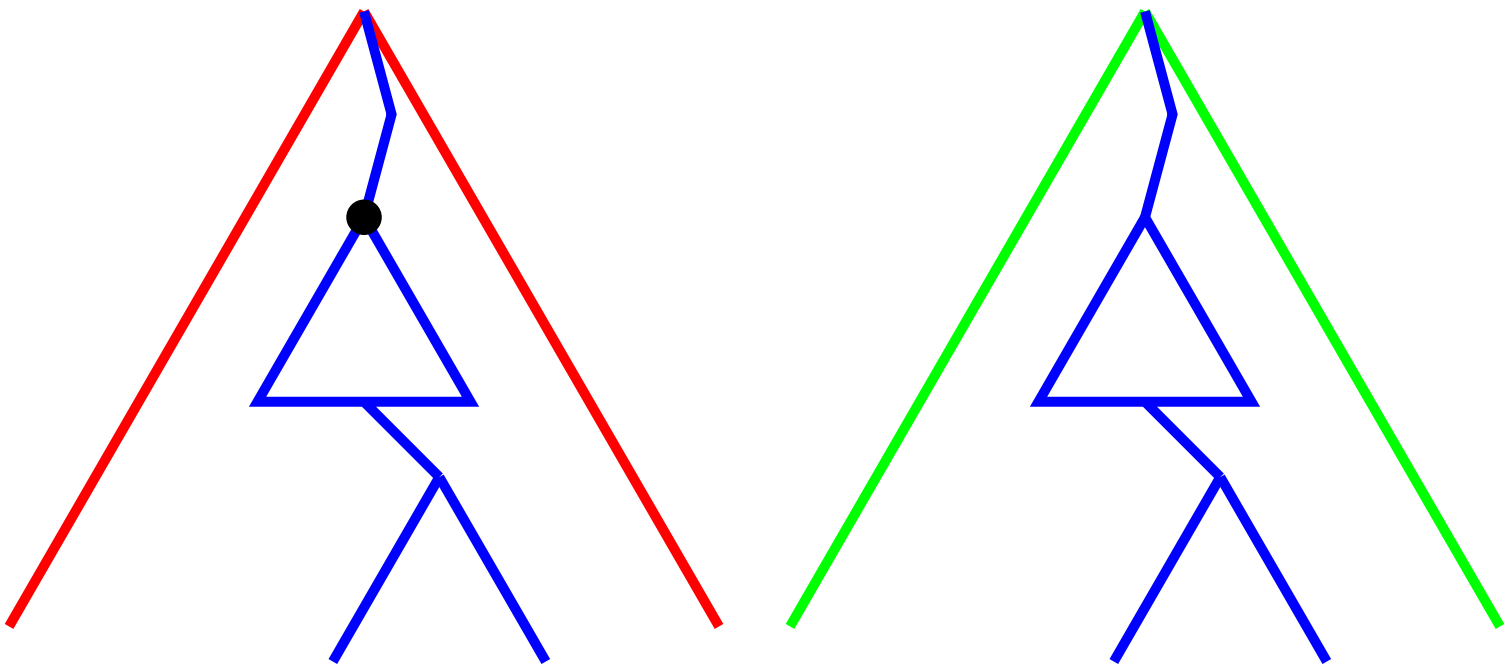
Конструкторы областей

- (! pin A
  - :a :b
  - (span :c :d)
  - :e
  - (span ””))
- (run update C …).



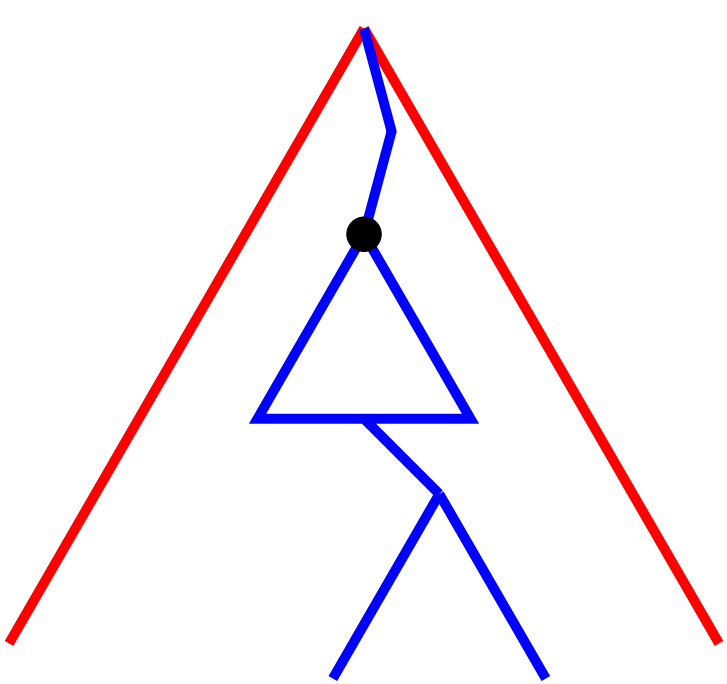
Конструкторы областей

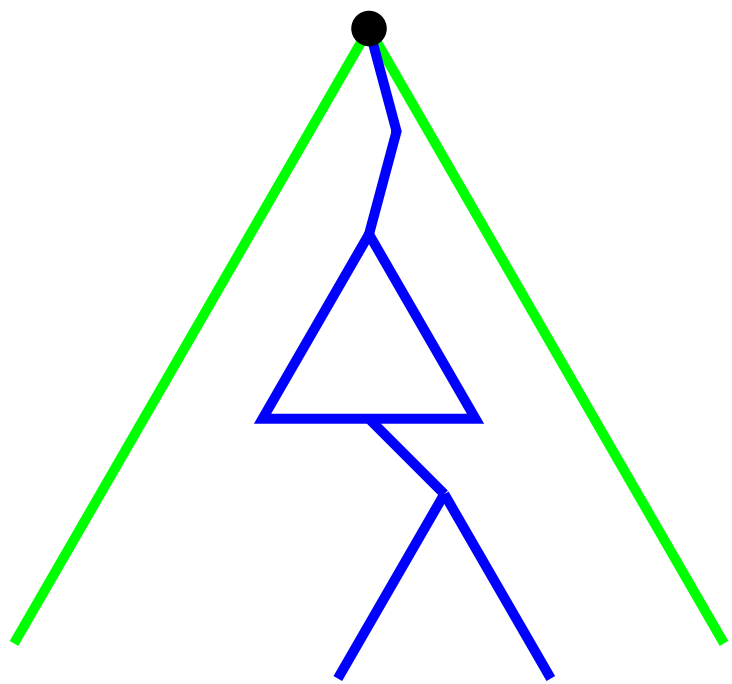
- (! pin A
  - :a :b
  - (span :c :d)
  - :e
  - (span ””))
- (run update C B …).



Конструкторы областей

- (! pin A
  - :a :b
  - (span :c :d)
  - :e
  - (span ””))
- (run update C B …).





Сложность и важность областей

- Вычислительно непростые префиксные деревья.
- Можно существенно упростить, ***специализировав*** под:
  - многомерные массивы с целочисленными индексами;
  - одномерные ассоциативные массивы;
  - записи с фиксированным набором полей.
- Области обеспечивают.
  1. Корректность потока данных:
     - простая проверка пересечения областей в новых продолжениях.
  2. ***Произведения*** для декартовой замкнутости.
  3. Сборку мусора:
     - семантика областей монотонная $\Longrightarrow$ сходимость;
     - всегда достаточно точно известно, что потребуется;
  4. «Бесплатные» следствия из сборки мусора:
     - обнаружение РАЙД-ошибок разрыва в потоке данных;
     - обработку ошибок в приложении;
     - контроль за ресурсами процессов.

- Рассчёты областей ***параллельны*** и ***локальны***.
- Сборка мусора ***нелокальный*** процесс, но:
  - ***асинхронный***
  - и ***фоновый***,
  - мир не нужно останавливать.

Процессы и руление

Процессы

- Дополнительный идентификатор для продолжений.
- Необходим для ручного завершения вычисления.
- Помогает изолировать ошибки: классика UNIX.

Руление

- `(run … (meta :cluster 2 :node 5 :ram 0 :gpu 1))`.
- `(! pin … (meta :check-point))`.

Плюсы и минусы в теории

+ РАЙД – это безграничный сетевой компьютер:
  - библиотеки распределённых функций;
  - по уровню сложности сравнимый с MapReduce;
  - небольшое число конструкций,
  - из которых можно попробовать сделать юниядро;
  - близкий к λ-исчислению язык:
    - почти обычный функциональный код;
    - ***возможна статическая проверка типов***.
+ Живучесть:
  - нет единых точек отказа;
  - возможность отслеживать и обрабатывать ошибки.
  - Всё рабочее состояние РАЙД – CRDT.
+ Производительность.
  - управляемость;
  - асинхронность;
  - данные в общей памяти можно не копировать;
  - децентрализованное хранение данных;
  - всё, кроме сборки мусора – независимо и локально;

Минусы и плюсы на практике

– Производительность:
  - работаем над этим;
  - опыт Hadoop внушает некоторый оптимизм.

\+ Живучесть:
  - проверена на прототипе.

- Выразительность:
  - проверена на компиляторе Си99 для МультиКлет;
  - поговорим об этом.

RiDE-2 в конструкторе компиляторов

Основные проблемы

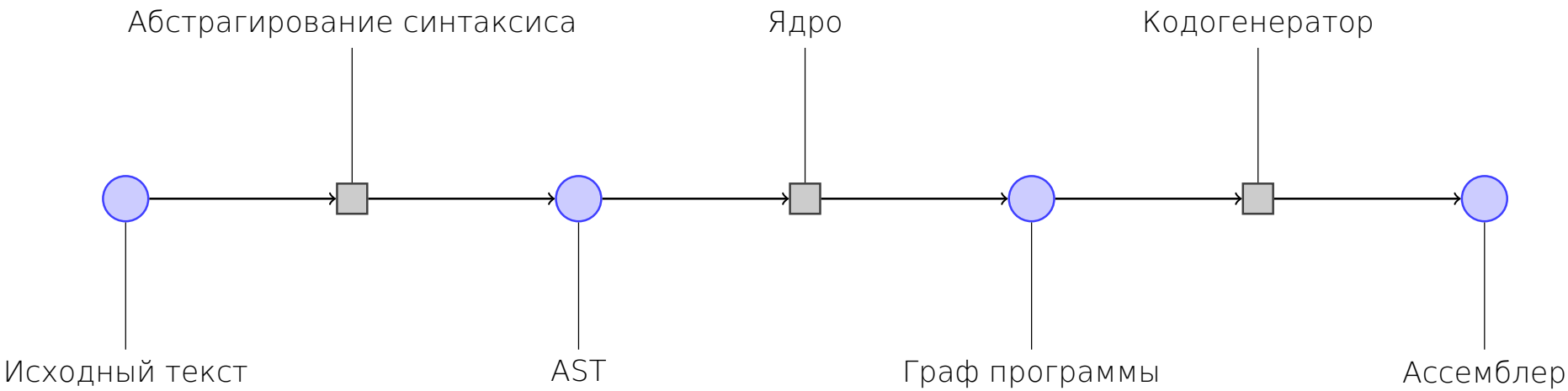
- Внутреннее представление машин слишком «регистровое».
  1. LLVM
  2. GCC
  3. LCC – не современный, но более «потоковый»; часто используют для экспериментов.
- Поддержка примитивов синхронизации, но без поддержки параллельных потоков управления.
  - У каждого линейного участка только один выход.
  - Ничто не мешает кодировать множественные передачи управления (модель RiDE).
- Слишком тяжёлый для модификации код:
  - Большой.
  - Много внутренних зависимостей.
  - Мало свободы в расширении языков.
  - Очень длинные циклы экспериментов.

} Дорого

Основные цели разработки

- Быстрые циклы разработки (***для продажи***).
  - Модульный код, не требующий вмешательства.
  - Простые и универсальные внешние интерфейсы между компонентами.
  - Распараллелить работу разработчиков.
- «Агностический» компилятор (***для продажи***).
  - От графов потока данных и потока управления не уйти.
  - Можно выводить графы, не фиксируя их конкретную форму.
  - Например, наши любимые множественные продолжения.
- Проверить модель RiDE (***настоящая цель***).
  - Модель параллельных вычислений в компиляторе!?
    - μ-типы похожи на процессы в модели CSP;
    - в общем-то, нет разницы граф какого именно потока выводить.

Схема конвейера




```
mcpp < source.c | mc-cfe | knl «список форм» | gen
```

Абстрагирование синтаксиса

- Исходный код → синтаксическое дерево.
  - Дерево двоичное для упрощения (?) ядра.
  - Дерево описывается операциями построения снизу:
    - сохранение порядка исходного кода;
    - понятная визуализация;
    - упрощение отладки.
  - Текстовое представление операций обеспечивает:
    - ручное вырезание куска синтаксиса для работы с ним;
    - эксперименты с синтаксисом в абстрактной форме;
    - упрощение отладки.
  - Операции – события для продолжений вывода графа.
  - Сама компиляция потоковая и параллельная.
- Быстрая параллельная коллективная разработка:
  - легко реализовать на основе bison, yacc, flex;
  - ↔ согласование семантики и синтаксиса;
  - перенос семантических конструкций между языками.

Пример абстрагирования синтаксиса

```
mcpp <<<'fn () { return a + b; }' | cfe

A 0.0 01.3."int"
L 0.4 06.11."@_func_decl"
A 0.0 02.2."fn"
L 0.3 06.6."@_argv"
A 0.0 ff.7."NOTHING"
E 0.0 06.6."@_argv" // 2
L 0.3 06.11."@_func_body"
U 0.0 0a.1."{"
A 0.2 01.6."return"
L 0.7 06.9."@retvalue"
A 0.0 02.1."a"
L 0.2 14.1."+"
A 0.2 02.1."b"
E 0.0 14.1."+" // 2
E 0.0 06.9."@retvalue" // 5
E 0.0 0a.1."{" // 8
E 0.0 06.11."@_func_body" // 10
E 0.0 06.11."@_func_decl" // 16
```

Ядро: основной компонент системы

- Трансформация AST в граф программы.
- Программируется в терминах форм:
  - срабатывающие по событиям продолжения графа программы;
  - полный аналог параллельных продолжений в RiDE.
- Обработку запускает внешний поток событий AST.
- Структура возникающих процессов соответствует AST.
- Интерфейсы между этими процессами простые и универсальные:
  - коллективная параллельная разработка семантики;
  - комфортный инкрементальный процесс;
  - возможна частичная реализация семантики, без контекста.
- Поддерживается концепция типов.
- Визуализация и инструментализация процесса вывода графа упрощает отладку.

Пример трансляции

```
knl -F ${rcc}/cfe/lib/lk/c-forms <<EOF
A 0.0 02.1."a"
L 0.2 14.1."+"
A 0.2 02.1."b"
E 0.0 14.1."+" // 2
EOF
```

```
(
  .L n0
  (
    .T     n0  ('00.1."P"; 4);
    .T     n1  ('00.1."I"; 4);
    .T     n2  ('00.5."world"; n0; n1);
    .TDef  n3  (n1; ('00.5."hello";
                     '00.4."test"; n2));
    .S     n4  ('02.1."a"; n1);
    .S     n5  ('02.1."b"; n1);
    .addr  n6  (n0; n4);
    .rd    n7  (n1; n6);
    .addr  n8  (n0; n5);
    .rd    n9  (n1; n8);
    .add   n10 (n1; n7; n9)
  );
  .Label n1 (01.2."L3"; n0)
)
```

Форма структуры бинарных операций

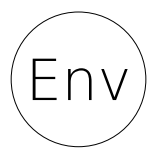
```
.FEnv (("L"; .T ('14.0."")); .F (
  .Nth op (.FIn (); (1));
  .FPut (0;
    (("rvalue"; "left"); ("rvalue"; "right"));
    .FEnv (("binop"; op)));

  .FPut (0; ("result"; "rvalue"); .F (
    .Nth position (.FIn (); (1; 1));
    .Nth result (.FIn (); (1; 0));
    .FOut (1; ((("rvalue"; position); result)));
    .FOut (.R (("UP")); ((("done"; position); 1)));
    .Done ()));

  .FPut (0; ("result"; "sequence"); .F (
    .Nth position (.FIn (); (1; 1));
    .FOut (.R (("UP")); ((("done"; position); 1)));
    .Done ()));

  .FPut (0; (("done"; "left")); .F (
    .FPut(0; (); .R (("LEFT")));
    .FPut(0; (("done"; "right"); ("rvalue"; "left")); .F (
      .FPut(0; (); .R (("RIGHT")))))));

  .Go (.E (("this")))))
```

S-выражения спешат на помощь!

```
(env-bind ("L" (type '14.0.""))
  (lambda tree-op op
    (run (env-search :binop op)
         (pin :rvalue :left) (pin :rvalue :right))

    (run (lambda result position
           (:= (! pin :rvalue position) result
               (! pin (area :UP) :done position) 1)
           (run done))
         (pin :result) (:pin sequence))

    (run (lambda resut postion (:= (! pin (area :UP) :done position) 1))
         (pin :result) (pin :sequence))

    (run (lambda (crop (area :LEFT))
                 (run (lambda (crop (area :RIGHT)))
                      (pin :done :right) (pin :rvalue :left)))
         (pin :done :left))

    (run go (env :this))))
```

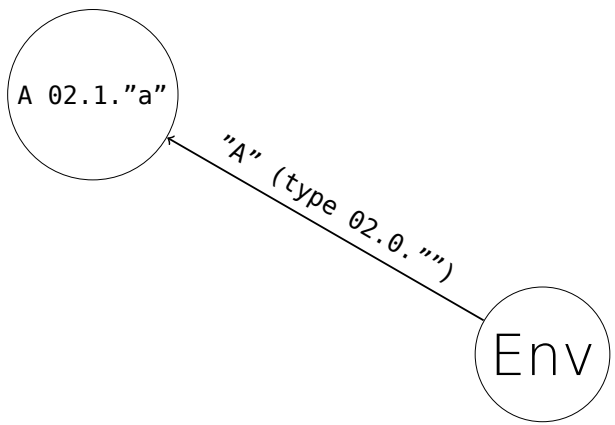
- В RiDE-2 области нелогичны: **`(area :UP)`** – ссылка.

Возникающая структура процессов



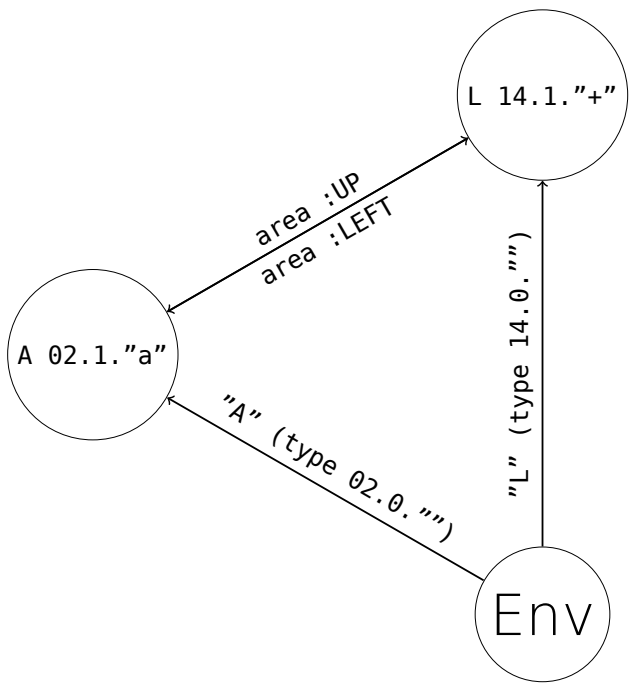
Возникающая структура процессов

- `A 0.0 02.1."a"`



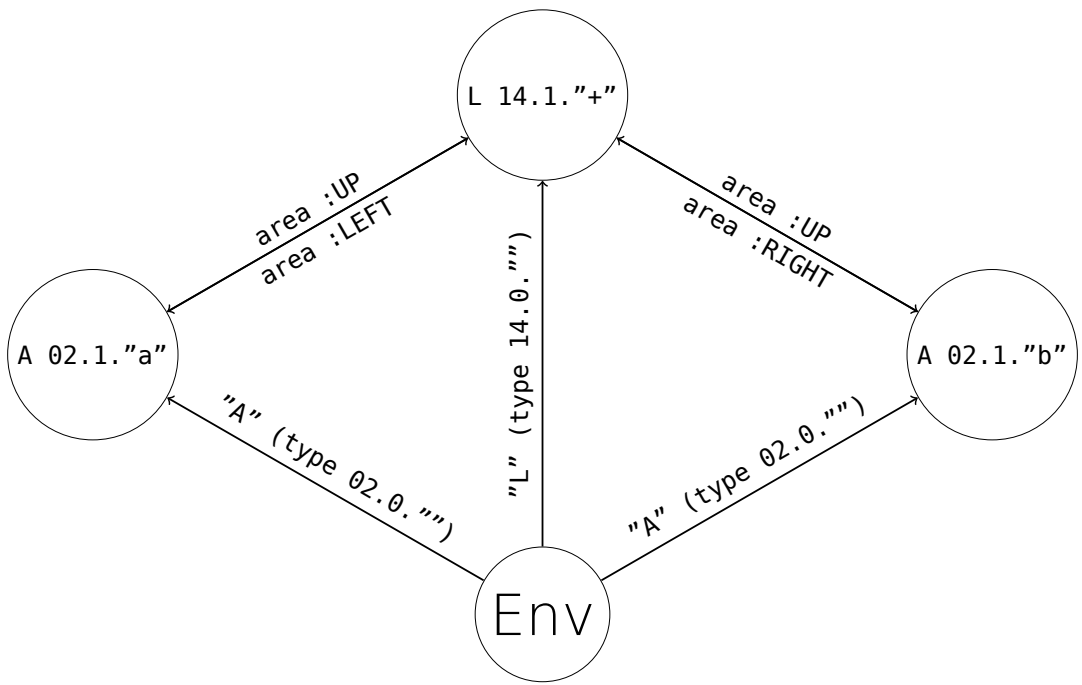
Возникающая структура процессов

- A 0.0 02.1.”a”
- L 0.2 14.1.”+”



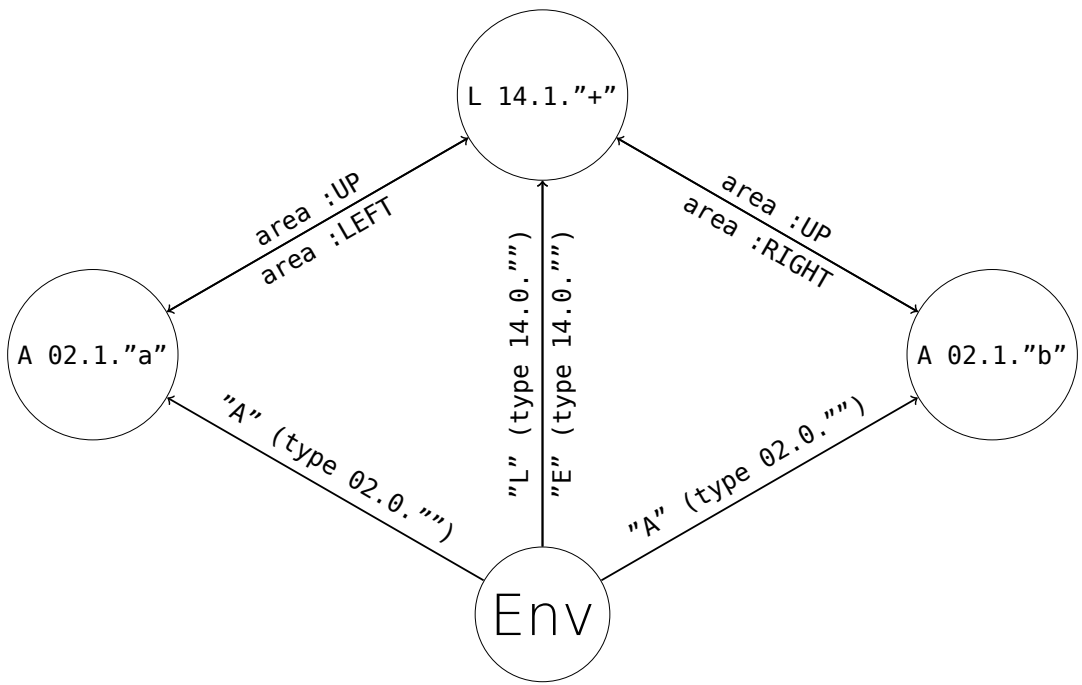
Возникающая структура процессов

- A 0.0 02.1.”a”
- L 0.2 14.1.”+”
- A 0.2 02.1.”b”



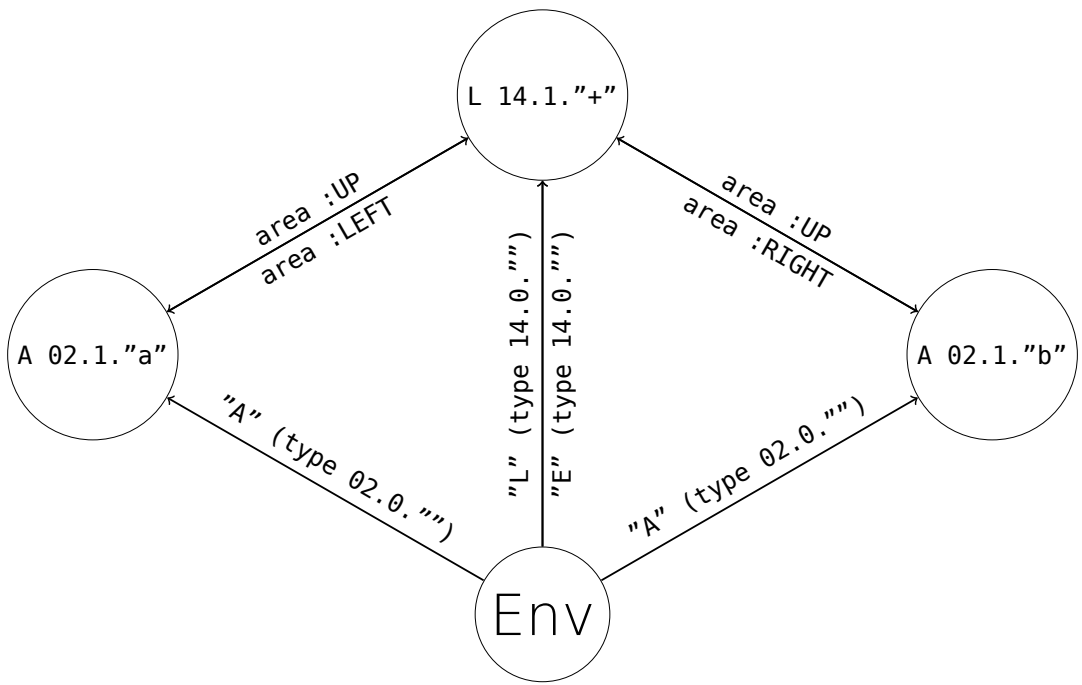
Возникающая структура процессов

- A 0.0 02.1.”a”
- L 0.2 14.1.”+”
- A 0.2 02.1.”b”
- E 0.0 14.1.”+”



Возникающая структура процессов

- `A 0.0 02.1.”a”`
- `L 0.2 14.1.”+”`
- `A 0.2 02.1.”b”`
- `E 0.0 14.1.”+”`
- Блоки данных – указатели на узлы графа.



Генератор кода (для C-99)

- В библиотеке есть необходимая поддержка:
  - загрузка и выгрузка графов;
  - типы;
  - символы;
  - метки и переходы.
- Стадии оптимизации могут быть отдельными программами.
  - Интерфейс - текстовое описание графа.
  - Программы могут быть на любом языке.
  - Чтобы получить грант: это даже могут быть DNN.
- Легко создавать тестовые примеры для отдельных синтаксических конструкций, поэтому разработка может быть коллективной и параллельной.

Плюсы

+ Циклы разработки короче. Больше экспериментов.
+ Параллельная разработка компонент:
  - коллективная, два человека могут работать над
    - `if (`e_1 `+` e_2`) {…}`,
    - *lvalue* `=` e_1 `+` e_2;
  - с инкрементальными согласованиями.
+ Инструментализация графов облегчает отладку.
+ Меньшая нагрузка на программиста.
+ Произвольная интерпретация графа программы:
  - экспериментальная система вёрстки;
  - визуализация процесса вывода – тоже интерпретация.
+ *RiDE работает*.

Минусы

- Требуется некоторый период обучения.
- «Дикий» синтаксис форм.
  - Работаем над этим.
- Двоичные AST – глупая и вредная идея.
  - Работаем над этим.
- Проверено на одном языке и одной архитектуре.

Дальнейшие работы

- Переход на более простую и выразительную RiDE-3.
  - Конструктор компиляторов – отличный полигон:
    - требуется ввод/вывод;
    - параллельные участки: построение графов операндов;
    - интенсивный обмен при построение графа оператора.
- Замена страшного синтаксиса на привычные sexp.
- Оптимизация циклов укрупнением линейных участков.
- Переход на произвольные AST.

Между прочим

- МультиКлет жив и развивается:
  - разработка защищённых средств связи;
  - разработка специализированных контроллеров;
  - новые версии ядер.
- И приглашает к сотрудничеству, если Вам интересны:
  - необычные процессоры,
  - странные компиляторы,
  - надёжные операционные системы реального времени.

Николай Викторович Стрельцов
n.streltsov@multiclet.com

Спасибо за внимание

m.bakhterev@imm.uran.ru