Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Реализация языка для обработки графов

Дипломная работа студента 544 группы
Крутихина Егора Александровича

Научный руководитель: Булычев Д. Ю.
Рецензент: Ломов Д. С.

Введение

- Огромное количество практических задач сводится к алгоритмам на графах
- Большинство задач требуют нестандартных решений (дополнительных эвристик)
- Высокоуровневые эвристики и сложные алгоритмы требуют от языка большой гибкости и выразительности
- Возникает потребность в специализированном языке, удобном для быстрого прототипирования алгоритмов

Постановка задачи

1. Разработать компилятор языка обработки графов GRAIL
2. Оценить выразительность языка, реализовав на нём некоторые алгоритмы и сопоставив с аналогами на других языках
3. Добиться приемлемой скорости работы программ, написанных на этом языке

Постановка задачи

1. Разработать компилятор языка обработки графов GRAIL
2. Оценить выразительность языка, реализовав на нём некоторые алгоритмы и сопоставив с аналогами на других языках
3. Добиться приемлемой скорости работы программ, написанных на этом языке

Постановка задачи

1. Разработать компилятор языка обработки графов GRAIL
2. Оценить выразительность языка, реализовав на нём некоторые алгоритмы и сопоставив с аналогами на других языках
3. Добиться приемлемой скорости работы программ, написанных на этом языке

Отличительные черты языка

- GRAIL — чистый функциональный язык с ленивой моделью вычислений
- Граф — базовая сущность языка, которая может передаваться в качестве параметра, участвовать в основных операциях и использоваться при построении более сложных структур данных (множеств графов, метаграфов и т.д.)
- Создание графов и запросы к графам — две основные синтаксические конструкции языка

Реализация

- Компилятор реализован на языке Objective Caml; код транслируется с GRAIL в язык Haskell
- Работы по оптимизации:
  1. выбор наилучшего внутреннего представления графов (библиотека времени выполнения на Haskell)
  2. выбор наилучших проекций для синтаксических конструкций языка (стадия генерации на Objective Caml)
- Интеграция с другими средами и библиотеками (сериализация/десериализация в формате DOT)

Пример

Поиск всех вхождений: Постановка задачи

Определение

Пусть $P = (V_P, E_P)$ *и* $S = (V_S, E_S)$ *— некоторые графы,* $M_V \subseteq V_P \times V_S$ *и* $M_E \subseteq E_P \times E_S$ *— отношения, заданные на вершинах и рёбрах этих графов.* *Вхождением графа* P *в граф* S *назовём пару инъективных отображений* $F_V : V_P \to V_S$ *и* $F_E : E_P \to E_S$ *таких, что:*

- *для любой вершины* $v \in V_P \quad (v, F_V(v)) \in M_V$;
- *для любого ребра* $e \in E_P$ *выполняются следующие соотношения:*

$$\begin{cases} (e, F_E(e)) \in M_E \\ src(F_E(e)) = F_V(src(e)) \\ dst(F_E(e)) = F_V(dst(e)) \end{cases}$$

Задача

Написать программу для нахождения всех возможных вхождений графа P *в граф* S *с отношениями* M_V *и* M_E.

Пример
Поиск всех вхождений: GRAIL

```
matchAll p s me mn = matchEdge ## ## (p{-e->}:e) where
  matchEdge corr i c =
    if c == {} then {(corr, i)}
    else
      let (e, c') = choose c in
      flatten {
       matchEdge (corr or #(src e)-()->(src f), (dst e)-()->(dst f)#) (i or #-!f->#) c' |
       f<-((s but i){-f->, _ when me e f && matchEnds e f}:f)
      }
    fi where
    matchEnds e f = matchEnd src && matchEnd dst where
     matchEnd r = mn (r e) (r f) && matched (r e) (r f);
     matched u v = (corr<w->:!v> ? label w == u : True) && (corr<:!u->w> ? label w == v : True);
    ;
  ;
;

convert g me mn =
  let degree   n = size (g{!n-e->} : e) + size (g{-e->!n} : e) in
  let nodeOrig n = size (g{m, _ when mn n m} : m) in
  let edgeOrig e = size (g{-f->, _ when me e f} : f) in
  let weight   n = (nodeOrig n) * (degree n) in
  let sqr      x = x * x in
  let cost     e = (edgeOrig e) * (sqr (weight (src e)) + sqr (weight (dst e))) in
  g[n] or #-!e-> | e<-[g{-e->}:e | \x y -> cost x <= cost y]#
```

Пример

Поиск всех вхождений: Haskell. Часть 1

```haskell
import Data.Map (Map)
import Data.IntMap (IntMap)
import Data.Graph.Inductive.Tree
import qualified Data.List as List
import qualified Data.Map as Map
import qualified Data.IntMap as IntMap
import qualified Data.Graph.Inductive.Graph as G

type L = String
type Graph = Gr L L
type Node = G.Node
type LNode = G.LNode L
type Edge = G.Edge
type LEdge = G.LEdge L
label g n = let Just l = G.lab g n in l

orderedEdges g me mn =
  let edges = G.labEdges g
      degree = G.deg g
      nodeOrig n = length $ filter (\(_,l) -> mn (label g n) l) (G.labNodes g)
      edgeOrig (_,_,l) = length $ filter (\(_,_,l') -> me l l') edges
      sqr x = x * x
      cost e@(src, dst, _) = (e, (edgeOrig e) * (sqr (nodeOrig src * degree src) +
                                                 sqr (nodeOrig dst * degree dst)))
  in map fst (List.sortBy (\ (_,d1) (_,d2) -> compare d2 d1) (map cost edges))
```

Пример

Поиск всех вхождений: Haskell. Часть 2

```
insertNode n to from | fst (G.match n to) == Nothing = G.insNode (n, label from n) to
insertNode _ to _ = to

insertEdge e@(src, dst, lab) to from =
  G.insEdge e (insertNode dst (insertNode src to from) from)

matchAll p s me mn es =
  matchEdges (Map.empty, IntMap.empty, G.empty, s, es, []) where
    matchEdges (eCorr, nCorr, newg, oldg, [], results) = (eCorr, nCorr, newg):results
    matchEdges (eCorr, nCorr, newg, oldg, pe@(pSrc, pDst, pLab):pes, results) =
      foldl f results (filter matchEdge (G.labEdges oldg)) where
        f results se@(sSrc, sDst, _) =
          matchEdges (eCorr', nCorr', newg', oldg', pes, results) where
            eCorr' = Map.insert pe se eCorr
            nCorr' = insNCorr pDst sDst $ insNCorr pSrc sSrc nCorr
            newg'  = insertEdge se newg oldg
            oldg'  = G.delLEdge se oldg
        insNCorr pn sn m = IntMap.insert pn sn $ IntMap.insert sn pn m
        matchNodes n1 n2 = case IntMap.lookup n1 nCorr of
                             Nothing  -> True
                             Just n2' -> n2 == n2'
        matchEdge (sSrc, sDst, sLab) =
          me pLab sLab &&
          mn (label p pSrc) (label s sSrc) && matchNodes sSrc pSrc && matchNodes pSrc sSrc &&
          mn (label p pDst) (label s sDst) && matchNodes sDst pDst && matchNodes pDst sDst
```

Пример

Поиск всех вхождений: тестирование производительности

- Граф P: 30 вершин, 34 ребра
- Граф S: 38 вершин, 74 ребра
- Общее число вхождений — 1280

	Скорость (сек.)	Память (МБ)
GRAIL	320	40
Haskell	30	30

Основные результаты

- Реализован компилятор для языка GRAIL в его текущем виде
- Разработан набор примеров на языке GRAIL
- Производительность доведена до некоторого «разумного» уровня: проигрыш аналогичным программам на Haskell — в 10 раз; улучшение производительности более чем в 100 раз по сравнению с первоначальным вариантом компилятора
- Подтверждена сила языка как средства для быстрого прототипирования: реализация алгоритма поиска всех вхождений на языке GRAIL позволила придумать и реализовать эвристику, улучшающую быстродействие алгоритма в 20 раз