#include "ip.h"

#include "display.h"

#include "constant.h"

#include <stack>

//Функция унификации двух предикатов

Lambda\* unification(FuncConstant\* a, FuncConstant\* b) {

Lambda\* l = new Lambda();

//Проверка равенства предикатов

if (a->getID() != b->getID()) return NULL;

if ( (typeid(\*a) == typeid(Predicate)) && ((Predicate\*) a)->isNegative() != ((Predicate\*) b)->isNegative()) return NULL;

if (a->getSize() != b->getSize()) return NULL;

for (int i = 0; i < a->getSize(); i++) {

Symbol\* ai = (\*a->getSymbols())[i];

Symbol\* bi = (\*b->getSymbols())[i];

//Если нет переменных и литералы не равны, то унификация невозможна

if ((typeid(\*ai) != typeid(Variable)) && (typeid(\*bi) != typeid(Variable)) && (!ai->cmp(\*bi))) {

return NULL;

}

if (typeid(\*bi) == typeid(Variable)) {

ai = bi;

bi = (\*a->getSymbols())[i];

}

if (typeid(\*ai) == typeid(Variable)) {

Symbol\* to = l->getReplace(ai);

// Если переменная присваивается 2 раза константе

if ((to != NULL) && (typeid(\*(to)) != typeid(Variable)) && (typeid(\*bi) != typeid(Variable)) ) {

delete l;

return NULL;

}

if (typeid(\*bi) != typeid(FuncConstant)) {

l->add(ai, bi);

}

else {

if (!(((FuncConstant\*)bi)->contain(ai))) {

l->add(ai, bi);

}

else {

return NULL;

}

}

}

else {

if (typeid(\*ai) == typeid(FuncConstant)) {

Lambda\* u = unification((FuncConstant\*)ai, (FuncConstant\*)bi);

if (u == NULL) return NULL;

l->extend(u);

}

}

// else {

// l->add(ai, bi);

// }

}

return l;

}

WType\* part\_divide(StatementLambda\* b, Divisor\* d) {

// std::vector<W\*>\* res = new std::vector<W\*>();

WType\* res = new WType();

res->q = 1;

res->b = \*b;

res->d = \*d;

int j = 0;

for (auto dP : \*d->getLiterals()) {

int i = 0;

for (auto bP : \*b->b->getLiterals()) {

W\* w = new W();

Lambda\* l = unification((FuncConstant\*) dP->literal, (FuncConstant\*) bP);

if (l != NULL) {

w->n = new StatementLambda();

w->n->b = b->b->replace(l, i);

w->n->l = b->l->copy();

w->n->l->extend(l);

w->d = d->replace(l, j);

// Display::getInstance()->printLine(w->n->b->toString(), 1);

res->n.push\_back(w);

if (w->n->b->getSize() == 0) {

Lambda\* lc = w->n->l->copy();

lc->extend(dP->lambda);

res->r.push\_back(lc);

}

if (w->n->b->getSize() > 0) {

res->q = g;

}

else {

if (res->q != g) res->q = 0;

}

}

i++;

}

j++;

}

return res;

}

Omega\* takeDivision(std::vector<StatementLambda\*> facts, StatementLambda\* D, Divisor\* d) {

Omega\* res = new Omega();

res->D = \*D;

res->q = 1;

// Первое частичное деление

WType\* firstDivide = part\_divide(D, d);

res->ws.push\_back(firstDivide);

// Добавление остатков в массив (для удобства)

for (W\* elem : firstDivide->n) {

res->n.push\_back(elem);

}

if (firstDivide->q != g) {

res->q = firstDivide->q;

return res;

}

std::stack<WType\*> vc;

for (W\* elem : firstDivide->n) {

// Выполнение второго шага полного деления

// Добавление инверсий литералов фактов

if (elem->n->b->getSize() > 0) {

for (StatementLambda\* s : facts) {

Predicate\* f = (\*s->b->getLiterals())[0]->copy();

f->setNegative(!f->isNegative());

elem->d->addLiteral(f, s->l);

}

WType\* a = part\_divide(elem->n, elem->d);

// Добавление остатков

for (W\* elem : a->n) {

res->n.push\_back(elem);

}

if (a->q == g) {

vc.push(a);

}

if (a->q == 0) {

res->q = 0;

for (Lambda\* rl : a->r) {

res->r.push\_back(rl);

}

}

res->ws.push\_back(a);

}

}

while (!vc.empty()) {

WType\* topElem = vc.top();

vc.pop();

for (W\* elem : topElem->n) {

// Выполнение второго шага полного деления

if (elem->n->b->getSize() > 0) {

WType\* a = part\_divide(elem->n, elem->d);

// Добавление остатков

for (W\* elem : a->n) {

res->n.push\_back(elem);

}

if (a->q == 0) {

res->q = 0;

for (Lambda\* rl : a->r) {

res->r.push\_back(rl);

}

}

if (a->q == g) {

vc.push(a);

}

res->ws.push\_back(a);

}

}

}

return res;

}

//bool isAbsorption(StatementLambda\* a, std::vector<StatementLambda\*>\* e) {

// for (StatementLambda\* elem: e) {

// if (a->b->getSize() != elem->b->getSize()) continue;

// for (int i = 0; i < a->b->getSize(); i++) {

// Predicate\* ai = a->b->getLiterals()[i];

// Predicate\* bi = elem->b->getLiterals()[i];

// if (ai->getID() != bi->getID()) break;

// if (ai->isNegative() != bi->isNegative()) break;

// }

// }

// return false;

//}

Step\* takeStep(std::vector<StatementLambda\*> D, Divisor\* divisor, Step\* parent, int ind) {

Step\* step = new Step();

step->q = 1;

if (parent != NULL) {

step->depth = parent->depth + 1;

step->c = parent->c;

}

else {

step->depth = 1;

}

for (d\* div: \*(divisor->getLiterals())) {

step->c.push\_back(new StatementLambda(div));

}

std::vector<StatementLambda\*> facts;

for (StatementLambda\* s : D) {

s->b->setLevel(step->depth, ind);

if (s->b->getSize() == 1) facts.push\_back(s);

}

std::vector<StatementLambda\*> e;

for (StatementLambda\* s: D) {

if (s->b->getSize() > 1) {

// Выполнение полного деления

Display::getInstance()->printLine(s->b->toString() + " // " + divisor->toString());

Omega\* omega = takeDivision(facts, s, divisor);

if (omega->q == 0) {

step->q = 0;

}

step->omegas.push\_back(omega);

for (W\* w: omega->n) {

if (w->n->b->getSize() > 0) {

Display::getInstance()->printLine(w->n->b->toString() + " " + w->n->l->toString(), 1);

bool eq = false;

for (StatementLambda\* c\_elem: step->c) {

if (c\_elem->b->isEqual(w->n->b)) {

eq = true;

break;

}

}

if (eq) continue;

e.push\_back(w->n);

step->c.push\_back(w->n);

}

else {

Display::getInstance()->printLine("0 " + w->n->l->toString(), 1);

}

}

}

}

std::vector<Divisor\*> divisors;

divisors.push\_back(new Divisor());

for (StatementLambda\* sl : e) {

// Расширяем количество выводимых высказываний

int divisorsSz = divisors.size();

for (int k = 0; k < (sl->b->getSize() - 1); k++) {

for (int i = 0; i < divisorsSz; i++) {

divisors.push\_back(divisors[i]->copy());

}

}

int i = 0;

for (Divisor\* elem: divisors) {

int k = i++ / divisorsSz;

Predicate\* newP = (\*sl->b->getLiterals())[k]->copy();

newP->setNegative(!newP->isNegative());

elem->addLiteral(newP, sl->l->copy());

}

}

// фиксирование результатов

step->d = divisor;

step->m = divisors;

step->p = 0;

if (e.size() == 0) {

step->p = 1;

}

return step;

}

Step \*conclusion(std::vector<StatementLambda\*> D, Divisor\* d, int depth, Step \*parent, int ind)

{

Step\* root = NULL;

if (parent == NULL) {

//Вызов процедуры шага

root = takeStep(D, d, NULL, 1);

root->depth = 1;

root->ind = ind;

}

else {

if (parent->depth == depth) {

return NULL;

}

root = takeStep(D, d, parent, ind);

}

if (root == NULL) return NULL;

int i = 1;

for (Divisor\* di : root->m) {

if (di->getLiterals()->size() < 1) continue;

Step\* si = conclusion(D, di, depth, root, i);

if (si != NULL)

root->sons.push\_back(si);

si = conclusion(root->c, di, depth, root, i);

if (si != NULL)

root->sons.push\_back(si);

i++;

}

return root;

}

std::vector<Lambda\*>\* reconciliation(Step\* root) {

std::vector<Lambda\*>\* v = new std::vector<Lambda\*>();

// Корневое решение не надо согласовывать

if (root->q == 0) {

for (Omega\* o: root->omegas) {

if (o->q == 0) {

for (Lambda\* lr : o->r) {

v->push\_back(lr);

}

}

}

}

std::vector<Step\*> sons = root->sons;

std::vector<Lambda\*> notRecon;

notRecon.push\_back(new Lambda());

// Находим всех потомков и формируем множество не согласованных подстановок

for (Step\* son : sons) {

for (Step\* stepSon: son->sons) {

sons.push\_back(stepSon);

}

if (son->q == 0) {

for (Omega\* o : son->omegas) {

if (o->q == 0) {

// Произведение множеств

int recSize = notRecon.size();

for (int j = 0; j < recSize; j++) {

for (int i = 0; i < o->r.size() - 1; i++) {

notRecon.push\_back(notRecon[j]->copy());

}

}

for (int i = 0; i < notRecon.size(); i++) {

notRecon[i]->extend(o->r[i / recSize]);

}

}

}

}

}

// Согласование

for (Lambda\* l : notRecon) {

bool isNotRecon = false;

std::vector<Replace\*>\* replaces = l->getList();

for (int i = 0; i < l->getSize(); i++) {

// Находим одинаковые переменные

Symbol\* from = (\*replaces)[i]->from;

std::vector<Symbol\*> unificationList;

unificationList.push\_back((\*replaces)[i]->to);

for (int j = l->getSize() - 1; j > i; j--) {

Symbol\* to = (\*replaces)[j]->from;

if ( (from->getID() == to->getID()) &&

(from->getLevel() == to->getLevel()) &&

(from->getIndex() == to->getIndex())) {

unificationList.push\_back((\*replaces)[j]->to);

(\*replaces).erase((\*replaces).begin() + j);

}

}

if (unificationList.size() < 2) continue;

// Пробуем унифицировать

// Проверка одинаковости типов

for (int j = 0; j < unificationList.size() - 1; j++) {

if (typeid(\*unificationList[j]) != typeid(\*unificationList[j+1])) {

isNotRecon = true;

break;

}

if ( (typeid(\*unificationList[j]) == typeid(Constant)) &&

(unificationList[j]->getID() != unificationList[j+1]->getID()) ) {

isNotRecon = true;

break;

}

}

if (isNotRecon) break;

// Унификация

if (typeid(\*unificationList[0]) == typeid(FuncConstant)) {

Lambda\* lam = unification((FuncConstant\*) unificationList[0], (FuncConstant\*) unificationList[1]);

if (lam != NULL) {

l->extend(lam);

}

else {

isNotRecon = true;

break;

}

}

}

if (!isNotRecon) {

// l->setReplace(replaces);

v->push\_back(l);

}

}

return v;

}