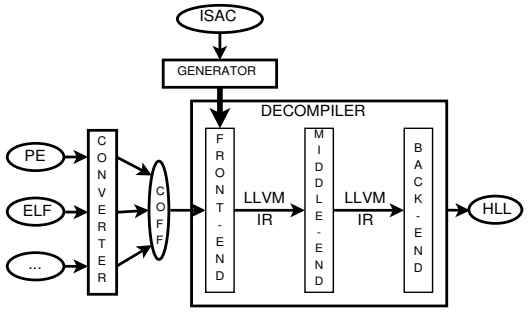
Rekonštrukcia datových typov pri spätnom preklade kódu

Autor: Peter Matula
Vedúci: Jakub Křoustek
Fakulta informačních technologií Vysokého učení technického v Brně

FIT VUT



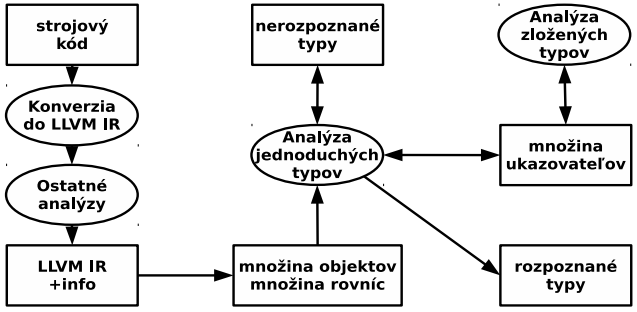
Spätný prekladač

Vstupný binárny program ⟶ Kód vo vyššom jazyku

- ⊕ Výstup čitateľnejší ako jazyk symbolických inštrukcií.
- ⊕ Rýchlejšia a jednoduchšia analýza programov.
- ⊕ Vhodné napríklad pre analýzu škodlivého kódu.
- ⊖ Extrémne náročná úloha, zatiaľ nedostatočne zvládnutá.

Rekonfigurovateľný spätný prekladač

- ⊕ Nezávislý na architektúre vstupu ⟶ preklad inštrukcií do medzikódu pomocou modelu CPU v jazyku ISAC.
- ⊕ Ako medzikód je použitý jazyk LLVM IR vytvorený v rámci projektu LLVM vyvíjajucom prekladač *clang* a ďalšie podporné nástroje.
- ⊕ Nezávislý na objektovom formáte vstupu ⟶ konverzia všetkých bežne používaných formátov do jednotného formátu COFF.
- ⊕ Nezávislý na použitom prekladači, optimalizácii, ...
- ⊖ Ešte náročnejšie ako jednoúčelový spätný prekladač.



Analýza datových typov

Datové typy sú dôležité pre vysokoúrovňové vyjadrenie programu.

Jednoduché typy

Jednoduché typy sú odvodené zo sémantiky jednotlivých inštrukcií. Využíva sa analýza toku dat rekonštruujúca typy objektov na základe propagačných rovníc a pravidiel zostavených pomocou princípov jazyka C.

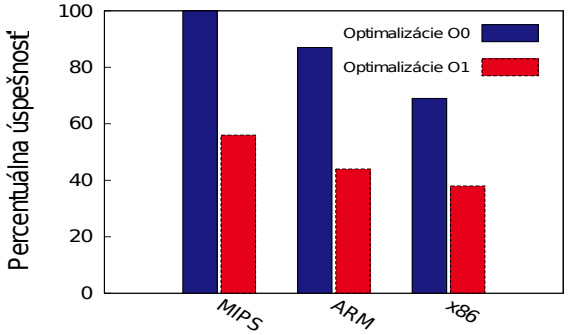
Typ objektu T je reprezentovaný trojicou množín vyjadrujúcich jeho jednotlivé atribúty:

$$T = \left(\tau^{core}, \tau^{size}, \tau^{sign}\right)$$

$$core \in \{int, ptr, float\},\ size \in \{1, 8, 16, 32, 64\},\ sign \in \{sign, unsign\}$$

Ku každej strojovej inštrukcii je vytvorená propagačná rovnica. Pre binárne sčítanie napríklad: $T_{dst} \Leftrightarrow T_{op1}\ ADD\ T_{op2}$. Množina rovníc je iteratívne riešená pomocou propagačných pravidiel vyvodzujúcich z typov operandov typ výsledku a naopak. Pravidlá sú špecifické pre každú operáciu. Napríklad pre binárne sčítanie (zlomková čiara značí implikáciu):

$$\frac{int \in \tau^{core}_{op1} \wedge int \in \tau^{core}_{op2}}{int \in \tau^{core}_{dst}},\ \frac{ptr \in \tau^{core}_{op1} \wedge int \in \tau^{core}_{op2}}{ptr \in \tau^{core}_{dst}},\ \frac{int \in \tau^{core}_{op1} \wedge ptr \in \tau^{core}_{op2}}{ptr \in \tau^{core}_{dst}}$$



Zložené typy

Zložené typy ako polia, štruktúry alebo únie sú kombináciou elementárnych a zložených typov. Na strojovej úrovni sa s prvkami pracuje jednotlivo. Agregované typy sú rozpoznané pomocou analýzy ofsetov použitých pre prístup do pamäte (analýza ukazovateľov).

Každému pamäťovému prístupu je priradený adresový výraz tvaru:

$$\left(base + offset\ + \sum_{j=0}^{n} C_j x_j\right)$$

Výraz vyjadruje spôsob, akým je vypočítaný ukazovateľ do pamäte. Hodnota $base$ značí základnú adresu prístupu – bázu. Môže sa jednať o skutočnú adresu (globálne premenné) alebo meno objektu, ktorý ju obsahuje (dynamicky alokované premenné). K báze relatívna hodnota $offset$ ďalej špecifikuje výslednú adresu. Posledný člen nazývaný multiplikatívna konštanta je typický pre iteratívny prístup k poliam. Nad adresovými výrazmi následne konštruujeme ekvivalenčné množiny reprezentujúce zložené objekty.

Pôvodný program

```
struct t {
  int t1, t2;
};
struct s {
  struct t a[4];
  float b[4];
};
struct s x;

int main(void) {
  for (int i=0; i<4; i++) {
    x.a[i].t1 = rand();
    x.a[i].t2 = rand();
    x.b[i] = rand();
  }
}
```

Spätný preklad bez analýzy

```
int main(int argc, char **argv)
{
  int32_t plum;
  int32_t orange;

  for (uint32_t i = 0; i < 4; i++)
  {
    plum = rand();
    orange = 8 * i;
    *(int32_t*)(orange + 143750160) = plum;
    *(int32_t*)(orange + 143750164) = rand();
    *(float32_t *)(4 * i + 143750192) =
    (float32_t)rand();
  }
}
```

Spätný preklad s analýzou

```
struct struct1 {
  uint32_t e0, e1;
};
struct struct2 {
  struct struct1 e0[4];
  float32_t e1[1];
};
struct struct2 apple = { .e1=0 }

int main(int argc, char **argv) {
  for (uint32_t i = 0; i < 4; i++) {
    apple.e0[i].e0 = rand();
    apple.e0[i].e1 = rand();
    apple.e1[i] = (float32_t)rand();
  }
}
```

Dosiahnuté výsledky

Pôvodný stav

V zadnej časti spätného prekladača boli bezprostredne odvodené jednoduché datové typy. Nedochádzalo ale k ich propagácii a interferencii. Zložené typy neboli rekonštruované vôbec.

Aktuálny stav

Odvodenie jednoduchých aj zložených datových typov v programoch určených pre architektúry MIPS, ARM a x86.
Úspešnosť na sade testovacích programov je zobrazená v grafe (96 testov). Praktická ukážka vplyvu na preklad na zdrojových kódoch.

Budúca práca

- Zvýšenie úspešnosti pri vyšších optimalizáciách.
- Optimalizácia časovej náročnosti algoritmu.
- Rekonštrukcia typov jazyka C++.

http://decompiler.fit.vutbr.cz/