ProCplusLog

Zápočtový program do C++

Martin Mareš

duben 2017

1 Základní popis programu

Program slouží jako jednoduchý interpret jazyka Prolog. V Prologu umožňuje provádět unifikace, pracovat s listy a používat prořezávání *(pomocí `!`)*. Program ale neumožňuje používat funkce nebo predikáty značící, že dvě proměnné se unifikují nebo neunifikují *například* `Var1 \= Var2` Program lze spustit pomocí příkazu:

```
$ ProCplusLog.exe [INPUT_FILE]
```

kde *[INPUT_FILE]* je povinný parametr označující vstupní soubor. Z tohoto souboru čte program pravidla, která slouží jako databáze faktů. Pokud program nedostane správný nebo validní vstup, tak zahlásí příslušnou chybu na standardní výstup. Program lze poté ukončit příkazem `halt.`.

2 Uživatelská dokumentace

2.1 Popis vstupu

Program přijímá termy ve standardní Prologové notaci. **Proměnné** začínají velkými písmeny, anonymní proměnné jsou označený `"_"` *(podtržítkem)*. **Složené termy** začínají malými písmeny a jejich argumenty jsou uzavřeny do kulatých závorek a odděleny čárkou. Pokud složený term nemá žádné argumenty je možné psát ho bez závorek a chápat ho jako atom - z pohledu aplikace jsou úplně totožné. V názvu proměnných, funktorů a atomů jsou dovoleny pouze alfanumerické znaky a podtržítko.

Jednotlivé složené termy lze spojovat do **pravidel**, pravidlo má hlavu ze složeného termu a volitelné tělo také ze složených termů. Pokud pravidlo nemá tělo jedná se o fakt. Hlava a tělo je oddělena posloupností `:-` *(dvojtečka pomlčka)*. V případě, že pravidlo nemá tělo je možné oddělovač vynechat. Konec pravidla se značí tečkou.

Je možné používat `%` pro **komentáře**. Pokud se na řádku vyskytuje znak procenta, tak zbytek řádku (včetně znaku procenta) je ignorován.

Zvláštním příkladem složených termů jsou **listy**. List je buď tvořený dvěma argumenty - hlavou a ocasem, nebo je prázdný. Při zápisu listu lze používat výčet `[prvni, druhy, treti]`. Výčet se vnitřně převádí do soustavy vnořených listů `[prvni|[druhy|[treti|[]]]`. Případně lze listy zápis pomocí dělaní na hlavu a ocas `[hlava| ocas]`. Případně lze tyto způsoby vzájemně kombinovat například `[prvni, druhy, treti| ocas]`, ocas může být vždy ale jen „jeden". List je vnitřně převoditelný na složený term s funktorem `[` a dvěma argumenty reprezentující hlavu a ocas - jedná se tedy pouze o syntaktické pozlátko.

Kromě těchto objektů lze požívat ještě řídící příkaz prologového **prořezávání** - `!`. Tento příkaz způsobí, že pokud bylo vykonávání aktuálního pravidla doposud úspěšné, tak už nedojde k hledání dalších možných pravidel v databázi a toto pravidlo bude poslední (pro příslušný rodičovský argument). Vnitřně je příkaz reprezentován jako složený term s funktorem `!`.

2.2 Popis ovládání programu

Program po spuštění načítá vstupní soubor s databází a zobrazí vstupní řádek na zadávání příkazů. Je možné zobrazit pravidla v databázi příkazem `list.` nebo program ukončit příkazem `halt.`. Zvláštním případem je příkaz `debug.`, který v knihovně aktivuje výpis informací ohledně postupu unifikace. Program pak vypisuje na standardní výstup pravidla, mezi kterými uvnitř probíhala unifikace a také její výsledek. Podle odsazení jednotlivých pravidle lze poznat, jak hluboko v zásobníku se daná unifikace prováděna.

V případě jiného textu program chápe vstup jako pravidlo pro dotaz. Platí stejná syntaxe jako u těla pravidla - argumenty jsou odděleny čárkou a konec je označen tečkou. Při zadání pravidla program hledá řešení a při nalezený řešení vypíše hodnoty unifikovaných proměnných, nebo vypíše *true/false*, pokud dotaz neobsahoval proměnné. Poté je možné pokračovat s hledáním dalšího řešení nebo hledání řešení ukončit příkazem `stop`. Program poté očekává vstup s novými dotazy (stejně jako na začátku).

3 Popis fungování programu

Program byl napsán v C++ za pomoci *Visual Studia 2017*, ale nevyužívá žádné specifické vlastnosti nebo knihovny, takže by měl fungovat i na jiných překladačích s novější verzí C++.

Program lze rozdělit na dvě části - na knihovnu prologu a na uživatelskou část. Samotná knihovna je samostatně funkční a umožňuje pracovat s termy a proměnnými pomocí objektů. Uživatelská část obsahuje parsovací knihovnu založenou na principu *SAX* a samotnou hlavní třídu, která se stará o interakci s uživatelem.

3.1 Popis knihovny

Knihovna má vstupní bod pro programátora v souboru "ProCplusLog.hpp"tento soubor vkládá ostatní části knihovny pomocí include direktivy překladače. Samotná knihovna je uložena ve vlastní složce "pro_cplus_log"a je dělena na 5 částí plus na jeden pomocný hlavičkový soubor. Jednotlivé objekty a funkce jsou napsané pomocí tempalatů, aby uživatel knihovny mohl volit typ funktorů a názvů proměnných podle jeho preferencí (například na něco, co se vypisuje stejně hezky jako *std::string*, ale porovnává se stejně rychle jako *int*).

3.1.1 Objekty

Základním stavebním kamenem je hlavičkový soubor "base.hpp". Ten obsahuje základní objekty pro uchovávání termů, složených termů a proměnných. Pro uživatele knihovny by měli být přístupné pouze tyto základní vnější objekty - samotná knihovna používá pro uchovávání dat vlastní objekty - modely. Term má vždy odkaz na svůj vnitřní model - díky tomu lze snadno provádět mělká kopie a lze také snadno zachovat standardní chování copy konstruktoru. Kde copy konstruktor vytváří novou (hlubokou) kopii - stejně jako například *std::string*. Vnější přístupný objekt je pouze skořápka, která sama

nic neobsahuje a schovává před uživatelem pointer na model. Z důvodu unifikace muselo být zvoleno pro uchovávání modelu *std::shared_ptru*. Na objektech jsou implementovány iterátory, které umožňují snadnější procházení. Zvláštní funkci má `variable_model`, který slouží pro uchovávání dat o proměnné. Má na sobě zároveň implementovanou hashovací funkci, který pomáhá při ukládání objektu do kontejneru (používá se jako vnitřní klíč při ukládání výsledků unifikace).

Pomocnou třídou je třída `rule`, která je v souboru "rule.hpp". Tato třída slouží pro spojování termů do pravidel, jako hlavu pravidla a argumenty jdou používat pouze složené termy (jako v klasickém prologu). Tato třída obsahuje iterátor nad tělem (argumenty pravidla) pro snadnější procházení.

V souboru "database.hpp"se nachází stejnojmenná třída `database`, která slouží pro uchovávání pravidel, každé pravidlo je uloženo v hashovací tabulce v *std::uniq_ptr*, kde jako hash se používá signatura funkce (například pro `foo(X, bla, Y).` je to `make_pair(foo, 3)`). Pravidlo jsou uložena v tabulce podle pořadí přidání, protože v Prologu záleží na pořadí pravidel. Třídu lze také procházet iterátorem nad pravidly.

3.1.2 Unifikace

Samotná unifikace se nachází v souboru "tools.hpp". V tomto souboru je metoda unify, která přijímá jako první dva parametry termy na unifikaci a jako třetí seznam výsledků - objekt `result_bindings`, do kterého se ukládají výsledky unifikace. Tento objekt umožňuje výsledky procházet, slučovat, vypisovat a měnit. Jednotlivé výsledky se ukládají do hashovací tabulky jako dvojice `variable_model` a ukazatel na `bind_model`. `Variable_model` je malý objekt, který obsahuje pouze unikátní číselné ID pro hashování nebo unifikaci (aby bylo možné od sebe odlišit termy v případě rekurzivních pravidel), a název proměnné, což se uchovává hlavně kvůli výpisům a ladění. `Bind_model` slouží hlavně pro spojování proměnných - v Prologu totiž může nastat situace, že dvě proměnné se unifikují do sebe a tyto unifikované proměnné se unifikují s jinými proměnnými do sebe a tak dále. Z toho důvodu `bind_model` ukládá seznam proměnných, které se s ním ztotožňují. Poté má pointer na `term_model` se kterým je unifikovaný. Díky tomu lze ukládat výsledky unifikace bez nutnosti dělat okamžitě kopie termů.

Důležitou metodou v `result_bindings` je metoda `merge`. Ta slučuje dva výsledky unifikace do sebe. V jedné verzi slučuje pouze neunifikované proměnné z původní třídy s těmi unifikovanými. V té druhé využívá seznam proměnných (typicky proměnné aktuálního pravidla) a u proměnných z tohoto seznamu kontroluje, jestli se neunifikovali dohromady (jestli nedošlo ke spojení dvou původně různých proměnných). Případně pak upravuje vnitřní reprezentaci dat. Výsledky unifikace lze procházet pomocí iterátorů.

Procházení databáze a hledání řešení je implementováno uvnitř třídy `solver`. Třída jako vstupní parametr přijímá pravidlo, jehož řešení se třída snaží najít. Hlava tohoto pravidla je ignorována. Při řešení postupně třída vyhledává pravidla v databázi a pokouší se unifikovat jejich hlavy s aktuálním argumentem aktuálního termu. Výsledky je poté možné procházet pomocí iterátorů. Třída má volitelnou možnost zobrazovat ladící informace, pak zobrazuje průběžně na standardní výstup, co se snaží unifikovat s čím.

Všechny kritické části knihovny jsou napsány nerekurzivně pomocí zásobníků (unifikace,

hledání řešení, slučování výsledků, kopírování, procházení apod...), rekurzivně jsou napsány pouze části, které pravděpodobně nebudou obsahovat velká data - výpis termu na výstup a čtení jednoho termu ze vstupu.

3.1.3 Proxy iterátor

Většina objektů knihovny využívá pro své fungování templaty `const_proxy_iterator` a `proxy_iterator` v souboru "proxy_iterator.hpp". Tyto třídy poskytují prostředek mezi vnitřními iterátory jednotlivých objektů a vnější reprezentací, která by měla být přístupná uživateli knihovny.

Protože iterátor konstruuje objekty nad modely, tak si iterátor musí uchovávat vnitřní hodnotu, kterou vrací na `opeprator->()`. Vzhledem k vlastnostem veřejně přístupných objektů si iterátor vlastně uchovává pouze ukazatel.

Templata přijímá jako templatový parametr typ kontejneru, nad kterým bude iterovat a třídu, která slouží pro překlad mezi vnitřním iterátorem a vnější reprezentací (hodnotou kterou bude vracet iterátor na `operator*()`.

3.1.4 Popis unifikačního algoritmu

Na vstupu je pravidlo, pro které chci najít řešení a mým cílem je získat toto řešení. Podívám se na první argument těla pravidla a v databázi najdu podle jeho signatury rozsah pravidel, která připadají v úvahu. Tyto rozsahy si uložím do zásobníku. Vezmu první term z rozsahu a udělám si jeho kopii (kvůli rekurzivním pravidlům a kvůli možnosti provádět změny v termu). Při kopírování si uložím, jak jsem měnil ID proměnných, abych v celém pravidlu postupoval stejně. Na kopii termu aplikuji změny podle výsledků unifikace z předchozího argumentu aktuálního pravidla (díky tomu nemusím kopírovat výsledky unifikace a můžu začít s prázdným objektem). Zkusím provést unifikaci mezi nalezeným pravidlem a aktuálním argumentem. Pokud neuspěje upravím rozsah pravidel v zásobníku. Pokud unifikace uspěje, tak si uložím její výsledek a pokračuji stejně prvním argumentem těla nalezeného pravidla.

Po čase se dostanu do stavu, kdy pravidlo nemá další argument, v takovém okamžiku sloužím výsledky unifikace s výsledky unifikace z rodičovského termu (použiji k tomu seznam změněných proměnných z rodičovského termu, takže nemusím kopírovat nezajímavé hodnoty). Poté pokračuji dalším argumentem rodičovského termu.

Může se mi také stát, že narazím na situaci, že nemůžu naleznout žádné vhodné pravidlo pro unifikaci s aktuálním argumentem. Pak to znamená, že je chyba nejspíše někde v předcích, takže mažu postupně zásobník než narazím na záznam u kterého lze ještě použít pro unifikaci jiné pravidlo z databáze.

Pokud v průběhu algoritmu narazím na prologové prořezávání `!`, tak upravím rozsah nalezených pravidel u rodiče, aby aktuální pravidlo bylo to poslední (posunu konečnou zarážku).

Hledání řešení končí úspěšně pokud narazím na poslední argument u pravidla, které nemá rodiče (jedná se vlastně o ono vstupní pravidlo). Hledání končí chybou, pokud si smažu celý zásobním (pokud nenacházím žádné unifikovatelné pravidlo).

3.2 Popis uživatelské části

Uživatelská část se skládá ze dvou částí - z parsovací třídy a z hlavní části, která interaguje s uživatelem. Parsovací třída funguje na podobném principu jako knihovny *SAX*. V konstruktoru dostává vstupní proud, který postupně čte. Dělí vlastně text na jednolité části prologové syntaxe a dovoluje uživateli třídy přečíst právě tu jednu část syntaxe. Případně dává možnost zjistit, před jakou částí syntaxe se uživatel třídy nachází. Zároveň třída přeskakuje všechny bílé znaky mezi jednotlivými částmi prologové syntaxe.

Jako volitelný parametr třída umožňuje zvolit si, jestli se přeskakují bílé znaky na konci každého příkazu nebo před začátkem každého příkazu. První metoda se hodí na čtení souboru, druhá je lepší na čtení uživatelského vstupu.

Hlavní program je uložen ve souboru "Prolog.cpp". Stará se o samotné vytváření objektů a volání funkcí knihovny. Při čtení se provádí i validování vstupu - testuje se, zda je dodržována správná syntaxe. V případě chyby je uživatel upozorněn a aplikace mu sdělí, co našla a co chtěla najít za syntaxi. Pro čtení zvláštních příkazů (vypsat databázi, konec apod...) je využito stejného mechanizmu jako při čtení pravidla na dotaz. Pouze se po přečtení pravidla aplikace pokouší unifikovat první argument dotazu s řídícím příkazem.