PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Implementace expertního systému v jazyce Common Lisp

2013 Jakub Kaláb

Anotace

Účelem expertního systému je asistovat expertovi na zvolenou problematiku, či jej plně nahradit. Expertní systém je schopen řešit širokou škálu problémů a odpovídat na dotazy na základě zadaných znalostí. Výsledkem práce je implementace prázdného expertního systému s dopředným i zpětným řetězením a syntaxí podobnou systému CLIPS. Ten je implementován jako knihovna v programovacím jazyku Common Lisp tak, aby jej bylo možno používat samostatně nebo integrovat do dalších programů.

Děkuji Mgr. Martinu Dostálovi, Ph.D. za vedení této diplomové práce.

Obsah

Seznam obrázků

Seznam příkladů

1. Úvod

Pojem expertního systému spadá do oblasti umělé inteligence. Jde o počítačový systém, který simuluje rozhodování experta nad zvolenou problémovou doménou. Expertní systém může experta zcela nahradit, nebo mu při rozhodování asistovat.

Ve své bakalářské práci [1] jsem implementoval základní knihovnu pro tvorbu expertních systémů (tzv. prázdný expertní systém) s dopředným řetězením v jazyce Common Lisp - ExiL[1]. Cílem této práce je knihovnu rozšířit o

- syntaktický režim pro zajištění přiměřené kompatibility se systémem CLIPS,
- možnost vrácení provedených změn včetně odvozovacích kroků,
- základní zpětné řetězení,
- podporu pro ladění s jednoduchým grafickým uživatelským rozhraním pro prostředí LispWorks™,
- podporu kompozitních podmínek (vnořené AND, OR a NOT) a jejich všeobecné kvantifikace.

Jazyk Common Lisp[2] (případně jiné dialekty Lispu) je častou volbou pro implementaci umělé inteligence díky svým schopnostem v oblasti symbolických výpočtů (manipulace symbolických výrazů), na nichž řešení těchto problémů často staví. Navíc jde o vysokoúrovňový, dynamicky typovaný jazyk, díky čemuž je programový kód stručný, snadno pochopitelný a tudíž jednoduše rozšiřitelný.

Syntax systému CLIPS[3] byla zvolena proto, že jde o reálně používaný systém[4], jehož syntax je Lispu velmi blízká, takže není těžké ji napodobit.

Expertní systém se syntaxí podobnou systému CLIPS implementuje v Common Lispu také projekt Lisa[5]. Ten však poskytuje pouze makra pro přímou práci se systémem nikoli funkce, které by bylo možné volat z jiného kódu. Tento nedostatek řeším pomocí funkčních alternativ maker popsaných v kapitole 3.3.10. Projekt také neposkytuje možnost zpětného řetězení, vracení provedených změn ani grafické uživatelské rozhraní.

V teoretické části textu popíši obecné principy expertních systémů, typické příklady jejich použití a způsoby, jakými expertní systém reprezentuje znalosti a vyvozuje z nich závěry. Stručně také představím syntaxi systému CLIPS a jeho možnosti.

[1] EXpert system In Lisp, původcem názvu knihovny je Zdenek Eichler

[2] http://en.wikipedia.org/wiki/Common_Lisp

[3] http://clipsrules.sourceforge.net

[4] http://clipsrules.sourceforge.net/FAQ.html#Q6

[5] http://lisa.sourceforge.net/

V praktické části poté popíši všechny možnosti knihovny ExiL, její instalaci a použití, architekturu programu, algoritmus RETE, na kterém implementace staví, a rozeberu některé aspekty implementace rozšíření.

2. Teoretická část

2.1. Expertní systém

Expertní systém je počítačový program, který usuzuje na základě znalostí z nějaké *problémové domény* za účelem vyřešení zadaného problému nebo poskytnutí odpovědi na otázku [2]. Problémovou doménou zde rozumíme množinu znalostí, omezenou na určitý okruh, který nás zajímá, s vyloučením všech ostatních, irelevantních znalostí [3]. Znalosti z problémové domény jsou reprezentovány jejím *modelem*[6]. Expertní systém tedy pracuje s modelem zvolené problémové domény.

K typickým problémům, řešeným pomocí expertních systému patří [2]

- plánování posloupnosti akcí vedoucích k zadanému cíli (např. plánování rozvrhu hodin se zohledněním obsazení učeben a vyučujících),
- diagnóza poruch systému a plánování jejich oprav, diagnóza a plánování léčby pacientů,
- konfigurace komplexních objektů (např. navržení serverového cloudu pro poskytnutí zadaných služeb),
- interpretace předem klasifikovaných dat (např. detekce nebezpečí z dat získaných z termografické kamery), analýza složení (vyhodnocení dat získaných metodami analytické chemie).

V první kapitole své bakalářské práce [1] uvádím typické vlastnosti expertních systémů, jejich zařazení v rámci informatiky a umělé inteligence a jejich rozdíly oproti jiným typům programů, které daný problém mohou řešit. Stěžejní charakteristikou expertního systému je úplné oddělení reprezentace znalostí, nad kterými systém usuzuje, od samotného *odvozovacího aparátu*. Odvozovací aparát je tedy obecný a je schopen usuzovat nad jakoukoli problémovou doménou, jejíž model lze v expertním systému reprezentovat.

Jak vyplývá z uvedené definice, expertní systém je hotový program schopný odpovídat na dotazy či řešit problémy v rámci zvolené problémové domény. Tento program můžeme vytvořit od základu v nějakém obecném programovacím jazyce. To je ale poměrně složité a pokud nemáme specifické nároky, nevyplatí se to. Častěji použijeme existující *nástroj* či *knihovnu pro tvorbu expertních systémů*. Tento nástroj už obsahuje odvozovací aparát a definuje reprezentaci znalostí a dotazů a její syntaxi (viz kapitola 2.2.). Poskytuje tedy tzv. *prázdný expertní systém*. Takovým nástrojem je i knihovna ExiL, která je výsledkem této práce.

[6] http://en.wikipedia.org/wiki/Domain_model

Zařazení expertního systému do procesu sestává z následujících fází:

1. sestavení modelu zvolené problémové domény,
2. výběr nástroje pro tvorbu expertních systémů podle typu domény a dotazů, které chceme být schopni zadávat,
3. reprezentace modelu problémové domény použitím syntaxe zvoleného nástroje.

Vytvořená reprezentace modelu problémové domény tvoří po načtení nástrojem pro tvorbu expertních systémů tzv. *znalostní bázi* výsledného expertního systému. Tomu pak můžeme zadávat dotazy buď přímo, použitím jeho syntaxe, nebo skrze nějaké uživatelské rozhraní. Je-li zvolený nástroj navržen jako knihovna, můžeme také vytvořit další program, který zadává znalosti do expertního systému a/nebo mu zadává dotazy a odpovědi na ně pak dále zpracovává.

Jednou z běžných vlastností expertních systémů je také schopnost vysvětlit, jak systém k řešení problému dospěl. To umožňuje evaluaci správnosti řešení a poskytuje tak zpětnou vazbu pro případnou opravu znalostní báze, zadaného cíle či dotazu, nebo odvozovacího aparátu (v případě vlastního návrhu expertního systému).

Expretní systém nelze použít k řešení jakéhokoli typu problému. To je dáno reprezentací znalostí v expertním systému a způsobem, jakým systém nad těmito znalostmi usuzuje. Vlastnosti znalostí, které lze v expertním systému reprezentovat, jejich reprezentace, způsoby zadávání cílů a dotazů i to, jak systém zadaných cílů dosahuje, je náplní následujících kapitol.

2.2. Reprezentace znalostí

Aby bylo možné pracovat se znalostmi v expertním systému, musí tyto splňovat několik vlastností [2]:

- přesnost - pro vyvození rozumných závěrů nemůže být znalost vágní (to neznamená, že nemůžeme vyjádřit nejistotu, i ta ale musí být definována přesně),
- relevance - znalosti by se měly týkat pouze pojmů z problémové domény
- organizovanost - znalosti by měly postihovat všechny potřebné spojitosti mezi pojmy (to zahrnuje například používání vždy stejného pojmu pro tutéž věc),
- uzavřenost - systém by pro vyvození závěrů neměl potřebovat žádné předchozí znalosti.

Systém nechápe význam znalostí, pracuje pouze s jejich reprezentací. Reprezentaci obecně chápeme jako množinu syntaktických a sémantických pravidel, která umožňují popsat nějaké entity [2]. Reprezentaci entity pak jako výraz, který podle těchto pravidel popisuje danou entitu. Jazyk reprezentace je množina všech možných reprezentací entit. V expertních systémech jsou těmito entitami právě znalosti.

Syntax reprezentace je množina pravidel, která definují základní stavební bloky (atomy) jazyka reprezentace a jak jejich kombinací vytvářet výrazy tohoto jazyka, tedy reprezentace jednotlivých entit [2]. Sémantika říká, jak jsou výrazy jazyka reprezentace interpretovány - jaký je jejich význam. Sémantika nám tedy dává zobrazení z reprezentace entity na reprezentovanou entitu. Sémantická pravidla jsou často rekurentní - přiřkneme význam atomům a poté jednotlivým strukturám, které vznikají jejich kombinací.

Aby byla reprezentace praktická, musí být [2]

- jednoznačná - není možné, aby měl jeden výraz jazyka reprezentace několik významů,
- expresivní - reprezentace musí umožňovat vyjádření všech potřebných detailů reprezentovaných entit,
- srozumitelná - význam výrazu by měl být snadno pochopitelný bez nutnosti chápat, jak je systémem zpracováván,
- stručná - výrazy by mělo být snadné psát, neměly by být zbytečně mnohomluvné;
- reprezentaci musí být možno zpracovávat a ukládat v počítači.

Míra expresivity je často v protikladu snadného strojového zpracování, hledáme tedy kompromis mezi těmito dvěma vlastnostmi.

Často používanými reprezentačními schematy jsou strukturované objekty, odvozovací pravidla a logické programy [2]. Knihovna ExiL, stejně jako systém CLIPS, používá prvních dvou. Typickou ukázkou třetího je jazyk Prolog[7].

Expertní systémy jsou *systémy založené na pravidlech*[8]. Jako takové rozlišují dva základní typy znalostí - *fakty* a *pravidla*. Fakt je elementární „statická“ znalost - vyjadřuje nějaké tvrzení z problémové domény, např. „obloha je jasná“. Odvozovací pravidlo je pak elementární odvozovací znalostí a má formu implikace. Pravidlo vyjadřuje, jaké (jaká) tvrzení můžeme odvodit z platnosti jiných tvrzení, např. „pokud je obloha jasná, neprší“. Pravidlo tedy sestává z *podmínek* („obloha je jasná“) a *důsledků* („neprší“).

[7] http://en.wikipedia.org/wiki/Prolog
[8] http://en.wikipedia.org/wiki/Rule-based_system

Stav systému je tvořen množinou aktuálně platných tvrzení a reprezentován množinou reprezentací odpovídajících faktů. Tvrzení tedy platí, pokud je reprezentace příslušného faktu součástí stavu systému. Počáteční stav expertního systému spolu s množinou definovaných odvozovacích pravidel tvoří tzv. *znalostní bázi* systému. Pokud se může množina definovaných pravidel v průběhu práce se systémem měnit, zařazujeme ji také do stavu systému. Ten je pak tvořen dvěma množinami - množinou faktů (tzv. *pracovní paměť*) a množinou definovaných pravidel (tzv. *produkční paměť*) [2].

Pojmy pracovní a produkční paměť nejsou příliš intuitivní. Jde o doslovný překlad v literatuře užívaných pojmů *working memory* a *production memory*. Pojem production memory je odvozen od označení *production rules*[9] pro odvozovací pravidla. Nejde ve skutečnosti o paměti, nýbrž o obsahy pomyslných pamětí. Pojmy „pracovní množina faktů“ a „pracovní množina pravidel“ by byly jistě výstižnější, bohužel ale také značně těžkopádné.

Různé expertní systémy se liší v reprezentaci faktů a pravidel. V různých expertních systémech tedy můžeme vyjádřit znalosti s různou mírou flexibility. Jednoduchým příkladem reprezentace faktů a pravidel je reprezentace prázného expertního systému STRIPS[10]. Atomy syntaxe této reprezentace jsou symboly. Ty reprezentují názvy jednotlivých objektů a relací z problémové domény. Fakty jsou pak ve tvaru `<relace>(<argumenty oddělené čárkami>)`, např. `At(robot,roomA)`.

V následujícím textu už nebudu rozlišovat mezi entitami a jejich reprezentací, pokud bude toto rozlišení jasné z kontextu, neboť to činí text zbytečně nepřehledným.

Pro vyšší flexibilitu pravidel definuje často reprezentace expertního systému speciální výraz, tzv. *vzor*. Ten má typicky stejnou strukturu jako fakt, ale může obsahovat speciální atomy - *proměnné*. Podmínky a důsledky pravidel jsou pak tvořeny nikoli fakty, nýbrž právě vzory. Díky tomu mohou být podmínky pravidla splněny mnoha různými posloupnostmi faktů.

Atomy proměnných začínají v reprezentaci STRIPSu velkými písmeny. STRIPSovým vzorem je tedy například `At(Box,Room)`. STRIPS umožňuje použití proměnných jen pro argumenty relace, samotný symbol názvu relace je neproměnný, navíc už velkým písmenem začíná (což je poněkud matoucí).

Pro potřeby vyhodnocování podmínek odvozovacích pravidel zavedu relaci *kongruence* faktu se vzorem (příp. faktu a vzoru, nebo i naopak). Fakt a vzor jsou kongruentní, pokud mají stejnou strukturu, tedy stejný název predikátu a stejný počet jeho argumentů, a pokud jsou neproměnné atomy vzoru stejné jako atomy na odpovídajících pozicích faktu. Vzor `At(Box,roomA)` je tedy kongruentní s faktem `At(box1,roomA)`, nikoli však s fakty `At(box1,roomB)`, `At(box1,roomA,floor2)` a `In(box1,roomA)`. Posloupnost faktů je kongruentní

[9] http://en.wikipedia.org/wiki/Production_system
[10] http://en.wikipedia.org/wiki/STRIPS

s posloupností vzorů, pokud mají tyto stejnou délku a jednotlivé fakty jsou kongruentní s odpovídajícími vzory (v daném pořadí).

Podmínky STRIPSového odvozovacího pravidla mohou být definovány například takto:

```
Preconditions: At(Box,Location), At(lift,Location), Level(Box,low).
```

Sémanticky jsou jednotlivé podmínky pravidla spojeny logickou konjunkcí a celá posloupnost je existenčně kvantifikována. Při vyhodnocování podmínek pravidla tedy hledáme (alespoň jednu) posloupnost faktů kongruentní s posloupností vzorů podmínek. Každý vzor podmínky je tedy spárován s jedním kongruentním faktem.

Konkrétní atom, který se ve faktu vyskytuje na pozici, kde je ve vzoru odpovídající podmínky proměnná, označujeme jako *vazbu* této proměnné. Při vyhodnocování podmínek je zajištěna konzistence těchto vazeb. Vyskytuje-li se táž proměnná na více místech ve vzorech podmínek pravidla, musí mít vždy stejnou vazbu. Pokud tedy najdeme posloupnost faktů kongruentní s podmínkami pravidla při zachování konzistence vazeb, říkáme, že podmínky pravidla jsou splněny touto posloupností faktů.

Předchozí podmínky jsou splněny například posloupností faktů `At(box1,z)`, `At(lift,z)` a `Level(box1,low)`. Použité vazby proměnných jsou `box1` pro `Box` a `z` pro `Location`. Kdyby ovšem byla v druhém faktu pozice `x`, nesplňovala by tato posloupnost uvedené podmínky, neboť by nebyla zachována konzistence vazeb.

Kdyby byly podmínky spojeny logickou disjunkcí, stačilo by nám nalézt jeden fakt kongruentní alespoň s jednou z podmínek. Kdyby byly podmínky naopak kvantifikovány všeobecně, znamenalo by to, že všechny posloupnosti faktů kongruentní s podmínkami pravidla musí zachovávat konzistenci vazeb proměnných. Některé expertní systémy umožňují definovat i takovéto podmínky pravidel.

Odvozovací pravidlo systému STRIPS je reprezentováno názvem, seznamem použitých proměnných, podmínkami a důsledky. Podmínky i důsledky pravidel jsou reprezentovány posloupnostmi vzorů. Před vzorem v důsledcích pravidla může být navíc použit speciální symbol `not`. Definice odvozovacího pravidla STRIPSu může tedy vypadat následovně [4]:

```
// move the box up a level
_MoveUp(Box,Location)_
Preconditions:  At(Box,Location), At(lift,Location), Level(Box,low)
Postconditions: Level(Box,high), not Level(Box,low).
```

Aplikace odvozovacího pravidla definuje přechod mezi dvěma stavy systému. Pravidlo je možné aplikovat, jsou-li splněny jeho podmínky. Při jeho aplikaci jsou vyhodnoceny jeho důsledky, na základě čehož je stav systému modifikován - reprezentace nějakých faktů jsou přidány do a/nebo odebrány z pracovní paměti systému.

Při aplikaci odvozovacího pravidla systému STRIPS jsou nejprve nahrazeny proměnné vzorů důsledků pravidla jejich vazbami získanými při vyhodnocování jeho podmínek. Tím převedeme vzory důsledků pravidla na fakty. Každý z těchto faktů je pak přidán do pracovní paměti, pokud není před vzorem odpovídajícího důsledku použit symbol `not`, nebo z paměti odebrán, pokud je tento symbol použit.

2.3. Systém CLIPS

Prázdný expertní systém CLIPS poskytuje obecnější a tudíž flexibilnější reprezentaci faktů a odvozovacích pravidel než systém STRIPS. Atomy faktů CLIPSu mohou být nejen symboly, ale také řetězce znaků a čísla (celá i desetinná). Systém rozlišuje dva typy faktů - jednoduché a strukturované. Jednoduchý fakt je reprezentován seznamem (uspořádanou n-ticí) atomů, např. `(in box A)`. Strukturovaný fakt je reprezentován objektem s pojmenovanými atributy (sloty), např. `(in (object box) (location A))`. Hodnotami slotů jsou opět atomy.

Vzory mají v CLIPSu stejný tvar jako fakty. Symboly proměnných zde začínají znakem `?`, např. `?location`. Kongruenci faktů a vzorů lze definovat podobně jako ve STRIPSu. U jednoduchých faktů srovnáváme atomy na odpovídajících pozicích, u strukturovaných pak hodnoty odpovídajících slotů objektu.

Systém CLIPS umožňuje spojovat podmínky odvozovacích pravidel logickými konjunkcemi i disjunkcemi a tyto libovolně vnořovat, navíc můžeme některé dílčí podmínky negovat. Dále je možné podmínky pravidel kvantifikovat jak existenčně, tak všeobecně. Podmínky CLIPSového odvozovacího pravidla tedy mohou vypadat například takto [5]:

```
(or (and (temp high)
         (valve closed))
    (and (temp low)
         (valve open))).
```

Definice pravidla CLIPSu vypadá například takto [5]:

```
(defrule system-flow
  (error-status ?status)
  (or (and (temp high)
           (valve closed))
      (and (temp low)
           (valve open)))
  =>
  (retract (error-status ?status))
  (assert (error-status confirmed))
  (printout t "The system is having a flow problem." crlf)).
```

Definice obsahuje název pravidla, jeho podmínky a důsledky. Podmínky jsou od důsledků odděleny symbolem =>. Důsledky pravidla nejsou pouze vzory faktů k přidání do či odebrání ze znalostní báze, nýbrž libovolné výrazy jazyka, který CLIPS definuje. Ty jsou při aktivaci pravidla, po nahrazení proměnných jejich vazbami, vyhodnoceny CLIPSovým interpreterem.

Systém CLIPS poskytuje velmi široké možnosti, další už nebudu uvádět. Lze je však najít v dokumentaci systému[11]. Praktická část této práce popisuje implementaci knihovny ExiL, která je výsledkem práce a implementuje prázdný expertní systém inspirovaný právě systémem CLIPS. Ve srovnání s ním však poskytuje jen velmi omezené možnosti. Knihovna je vytvořena v programovacím jazyce Common Lisp a v důsledcích pravidla lze použít libovolné lispové výrazy, včetně volání maker pro modifikaci stavu systému.

2.4. Inference

V přechozích kapitolách jsem definoval stav expertního systému a aplikaci odvozovacího pravidla jako přechod mezi dvěma stavy. Výpočet expertního systému, tedy inferenci, můžeme definovat jako posloupnost stavů systému, mezi nimiž systém přechází aplikacemi odvozovacích pravidel. Výpočet ovšem může probíhat dvěma různými směry.

Expertní systém s dopředným řetězením (inferencí) vychází při výpočtu z výchozího stavu systému, tedy ze stavu, kdy jsou v pracovní paměti fakty ze znalostní báze. Poté v krocích vyhodnocuje, která odvozovací pravidla mají splněné podmínky a které posloupnosti faktů z pracovní paměti je splňují. Z těchto dvojic (pravidlo, posloupnost faktů), zvaných shody (*match*), pak systém jednu vybere a aktivuje pravidlo s použitím vazeb proměnných získaných navázáním vzorů podmínek na odpovídající fakty. Takto systém postupuje, dokud není inference zastavena, nebo už neexistuje žádná další shoda.

Takovým systémem je i CLIPS. Výsledkem výpočtu systému je tedy určitý stav, ve kterém se inference zastavila, spolu s posloupností použitých shod. K ovlivnění průběhu inference, tedy výběru shody, která bude v daném kroku aplikována, slouží inferenční strategie. Ty definují uspořádání množiny shod, ze kterých je pak vybrána první. Několik inferenčních strategií popíšu v kapitole 3.3.4.

Expertní systém se zpětným řetězením (inferencí) naopak umožňuje definici cílů, kterých chceme dosáhnout. Systém poté hledá pravidla, která vedou ke splnění jednotlivých cílů, zkoumáním jejich důsledků. Takovýto způsob výpočtu nazýváme také v oblasti umělé inteligence jako *means-end analýza*[12]. Výhodou zpětného řetězení je právě možnost zadání konkrétních cílů, jejich implementace je ale náročnější a proto tyto systémy často poskytují omezenější možnosti co do

[11] http://clipsrules.sourceforge.net/OnlineDocs.html
[12] http://en.wikipedia.org/wiki/Means-ends_analysis

definice odvozovacích pravidel. Systém CLIPS například umožňuje použití libovolných volání jeho jazyka v důsledkové části pravidel. Důsledky těchto volání ale nelze obecně předvídat, systém se zpětným řetězením tedy takovou možnost poskytovat nemůže.

Systém STRIPS používá zpětné inference. Pravidla se tedy ve skutečnosti neaplikují tak, jak jsem popsal v kapitole 2.2. Přesto o nich z uživatelského hlediska můžeme takto uvažovat. Tato představa je pro konstrukci pravidel přímočařejší, navíc nás posloupnost aplikovaných pravidel většinou zajímá v pořadí jejich pomyslné aplikace, tedy proti směru inference.

Cíle v systému STRIPS zadáváme ve formě vzorů, stejně jako podmínky pravidel. Pokud systém zvolí k aktivaci nějaké odvozovací pravidlo, protože jeho aktivace vede ke splnění aktuálního cíle, přidá podmínky tohoto pravidla do množiny cílů. Navázání proměnných funguje při aplikaci pravidla opačně, než jsem u aplikace pravidla popsal.

V případě splnění všech cílů, zadaných uživatelem i odvozených při aplikaci pravidel, je výsledkem výpočtu posloupnost pravidel, které je třeba aplikovat, abychom přešli z počátečního stavu systému do cílového. Pokud obsahovaly zadané cíle proměnné, je součástí odpovědi také množina použitých vazeb těchto proměnných, tedy jejich *substituce*.

U expertních systémů s dopředným řetězením můžeme také definovat cíle výpočtu. Tyto ale musíme přidat jako fakta do znalostní báze a v pravidlech je musíme explicitně testovat, případně modifikovat, což může být velmi složité.

Budeme-li uvažovat všechny možné cesty, kterými se systém může v každém stavu ubírat, ať už aplikací splněného pravidla, nebo pravidla, jehož důsledky vedou ke splnění cíle, můžeme dosažitelné stavy uspořádat do stromu. Ten bude mít kořen buď v počátečním stavu (u systému s dopředným), nebo v cílovém (u systému se zpětným řetězením).

Výpočet si tedy můžeme představit také jako prohledávání tohoto stromu stavů. Protože množství dosažitelných stavů je často obrovské (narůstá exponenciálně vzhledem k počtu kroků výpočtu), systém neprohledává celý strom, nýbrž používá heuristik k rozhodnutí, kterou větví výpočtu se ubírat. Těmito heuristikami jsou právě zmíněné inferenční strategie.

Přiložená knihovna ExiL implementuje jak dopřednou, tak omezenou formu zpětné inference. Její použití a implementaci popisuje praktická část práce.

3. Praktická část

Tato kapitola popisuje knihovnu ExiL, která je výsledkem této práce. Nejprve popíšu její instalaci a prerekvizity nutné k jejímu používání. Pak v uživatelské příručce představím její možnosti a typickou strukturu programu, který ji využívá. Poté v referenční příručce shrnu všechny možnosti, které knihovna poskytuje. Nakonec v části věnované implementaci popíšu architekturu zdrojového kódu knihovny a algoritmus RETE sloužící k efektivnímu vyhodnocování podmínek pravidel a zmíním zajímavé části kódu implementující jednotlivá rozšíření.

3.1. Potřebná znalost Common Lispu

Knihovna ExiL je implementována v jazyce Common Lisp. Pro práci s ní a pochopení její implementace je tedy potřebná znalost tohoto jazyka. Za perfektní příručku k jazku Common Lisp považuji knihu Practical Common Lisp[13], která je k dispozici online[14]. Následující seznam uvádí relevantní pojmy z Common Lispu a kapitoly z knihy, kde se o nich lze dočíst. V dalším textu budu vždy první výskyt nového pojmu značit kurzívou.

2. Lather, Rinse, Repeat: A Tour of the REPL
implementace lispu, Lispbox, práce s REPLem, obsluha výjimek

4. Syntax and Semantics
prefixová syntax, S-výrazy

5. Functions
funkce - definice, typy parametrů, anonymní funkce

6. Variables
proměnné - vytváření vazeb, lexikální proměnné a uzávěry, dynamické proměnné, přiřazení, makra upravující hodnoty proměnných

7. Macros: Standard Control Constructs
řídicí struktury, cykly

8. Macros: Defining Your Own
makra - jejich funkce, vyhodnocování, definice, parametry

10. Numbers, Characters, and Strings
základní vestavěné typy a operace s nimi

11. Collections
kolekce (pole, hashovací tabulky) a práce s nimi

[13] Siebel, P.: *Practical Common Lisp*. Apress, 2005, ISBN 1-59059-239-5.
[14] http://www.gigamonkeys.com/book/

12. They Called It LISP for a Reason: List Processing
seznamy - reprezentace, operace

13. Beyond Lists: Other Uses for Cons Cells
stromy, množiny, asociativní a property listy

16. Object Reorientation: Generic Functions
CLOS[15] - generické funkce, metody, multimetody, řetězec volání

17. Object Reorientation: Classes
CLOS - třídy a objekty, definice, sloty, dědičnost

19. Beyond Exception Handling: Conditions and Restarts
práce s výjimkami

20. The Special Operators
lexikální a top-level prostředí, lokální funkce a proměnné; kontrola vyhodnocování S-výrazů, kvotování; načítání zdrojových souborů pomocí `load`

21. Programming in the Large: Packages and Symbols
package a práce se symboly - klíčky, import, export, zastiňování

Jako referenční příručku vestavěných funkcí a maker Common Lispu pak lze použít Common Lisp HyperSpec™[16].

[15] http://en.wikipedia.org/wiki/Common_Lisp_Object_System
[16] http://www.lispworks.com/documentation/HyperSpec/Front/

3.2. Instalace

3.2.1. Získání zdrojového kódu

Zdrojový kód knihovny ExiL je přiložen k této diplomové práci a lze jej také získat zklonováním[17] gitového[18] repozitáře na adrese `git@github.com:Incanus3/ExiL.git`. Kód knihovny ja na přiloženém CD v adresáři `src`, umístění zdrojových souborů budu vztahovat vzhledem k tomuto adresáři.

3.2.2. Prerekvizity

Pro práci s knihovnou ExiL potřebujeme lispový interpreter[19], vývojové prostředí (s interpreterem bychom si ve skutečnosti vystačili, ale přímá práce s ním není většinou příliš pohodlná) a knihovny umožňující dávkové načtení celého projektu včetně závislostí.

Knihovnu jsem vyvíjel v prostředí SLIME[20], což je plugin pro textový editor GNU Emacs[21] (poskytující mimo jiné pomůcky pro editaci lispového zdrojového kódu, REPL[22] a debugger), s interpreterem SBCL[23] a tuto kombinaci mohu vřele doporučit. V operačním systému Debian GNU Linux, který jsem pro vývoj použil, lze Emacs, SLIME i SBCL nainstalovat z výchozího repozitáře a aktivovat úpravou inicializačního souboru Emacsu[24]. Prostředí poté můžeme v Emacsu spustit voláním příkazu `slime` (`<Alt+X>slime<ENTER>`). Při prvním spuštění se kód prostředí kompiluje, což může chvíli trvat, pak už se v editoru otevře buffer[25] s lispovým REPLem.

Knihovnu jsem testoval také ve vývojovém prostředí LispWorks™[26] Personal Edition 6.1, pro které jsem také vytvořil minimalistické grafické uživatelské rozhraní. Součástí prostředí LispWorks je i lispový interpret. Prostředí můžeme získat na adrese `http://www.lispworks.com/downloads/index.html` a nainstalovat podle návodu, který se zobrazí po vyplnění formuláře.

Zběžně jsem knihovnu testoval také v prostředí Lispbox[27], což je balík umožňující hormadnou instalaci editoru Emacs, prostředí SLIME a interpreteru Clo-

[17] `http://git-scm.com/docs/git-clone`
[18] `http://git-scm.com/`
[19] lispové interpretery jsou většinou zároveň kompilátory, označením interpreter tedy budu nazývat obojí
[20] `http://www.common-lisp.net/project/slime/`
[21] `http://www.gnu.org/software/emacs/`
[22] `http://en.wikipedia.org/wiki/Read-eval-print_loop`
[23] `http://www.sbcl.org/`
[24] `http://www.common-lisp.net/projects/slime/doc/html/Installation.html`
[25] `http://www.gnu.org/software/emacs/manual/html_node/emacs/Buffers.html`
[26] `http://www.lispworks.com/`
[27] `http://common-lisp.net/project/lispbox/`

zure CL[28] dostupný pro Windows, GNU Linux i Mac OS X.

Pro efektivní načtení knihovny včetně závislostí potřebujeme ještě dvě knihovny:

- ASDF[29] je knihovna umožňující snadnou definici struktury projektu a jeho dávkové načtení,
- quicklisp[30] staví na knihovně ASDF a umožňuje pohodlně stáhnout a načíst knihovny třetích stran z internetové databáze.

Knihovna ASDF je součástí instalace interpreteru SBCL i prostředí LispWorks. Knihovnu quicklisp jsem k projektu přiložil a pokud není součástí prostředí, je automaticky načtena před načtením ExiLu.

3.2.3. Načtení knihovny

V prostředí SLIME (interpreter SBCL) načteme knihovnu načtením souboru `load.lisp` z kořenového adresáře knihovny, tedy zadáním

```
(load "cesta/k/projektu/src/load.lisp")
```

v REPLu). Tento soubor nejprve načte knihovnu `quicklisp`, je-li potřeba, a s její pomocí poté načte celý projekt ExiL včetně závislostí. Pro prvotní stažení závislostí je třeba internetového připojení. Nakonec soubor definuje výchozí prostředí, viz kapitola 3.3.9.

V prostředích LispWorks a Lispbox (Clozure CL) načítání pomocí knihovny `quicklisp` nefunguje správně, knihovnu je proto třeba načítat pomocí souboru `load-manual.lisp`. Načíst můžeme opět voláním `load` v REPLu, nebo vybráním položky `Load...` v nabídce `File` menu libovolného okna prostředí LispWorks.

Všechna *makra* a *funkce*, které knihovna definuje pro přímé volání, jsou *exportována* z *package* `exil`. Před interakcí s knihovnou je tedy třeba vstoupit do package `exil-user`, který *symboly* z package `exil` *importuje*, voláním `(in-package :exil-user)`. Symboly z package je také možno importovat do existujícího package takto:

```
(defpackage :my-package
  (:documentation "user-defined package")
  (:use :common-lisp :exil)
  (:shadowing-import-from :exil :assert :step))
```

Package `exil` exportuje několik symbolů, které již v package `common-lisp` existují. Ty je třeba *zastínit*, jak je vidět z ukázky.

[28] http://www.clozure.com/clozurecl.html
[29] http://common-lisp.net/project/asdf
[30] http://www.quicklisp.org/beta/

3.3. Uživatelská příručka

3.3.1. Struktura programu

Příklad 1 na straně 21 ukazuje minimální strukturu programu nad knihovnou ExiL (dále exilový program). První část programu tvoří definice znalostní báze. Ta sestává z definic faktů, ze kterých expertní systém vychází, a definic odvozovacích pravidel, jež jsou následně aplikována při inferenci.

Definice faktů jsou uspořádány do skupin označených názvem (v tomto případě `world`). V ukázkovém programu si snadno vystačíme s jednou skupinou faktů, v reálných programech bude ale těchto skupin většinou více. Tato organizace umožňuje snadnou redefinici, případně odebrání, jen některých skupin faktů v případě potřeby. Definice skupiny faktů `world` v příkladu přidává do znalostní báze informaci o počáteční pozici robota, krabice a o našem záměru přesunout krabici z pozice `A` na pozici `B`.

Následuje definice odvozovacích pravidel. Definice každého pravidla sestává z jeho názvu, volitelného řetězce sloužícího k dokumentaci pravidla, množiny podmínek, tedy předpokladů pro jeho splnění (a následnou aktivaci), a množiny důsledků, tedy libovolných lispových výrazů, které jsou při aktivaci pravidla vyhodnoceny. Tyto dvě množiny jsou od sebe odděleny symbolem `=>`.

Podmínky odvozovacích pravidel jsou ve formě vzorů. Struktura vzorů je stejná jako struktura faktů (viz kapitola 3.3.2.), ale na rozdíl od nich mohou obsahovat proměnné (symboly začínající otazníkem). Při vyhodnocování podmínek pravidla je zajišťěna konzistence vazeb těchto proměnných a výskyty všech proměnných v důsledcích pravidla jsou při jeho aktivaci nahrazeny jejich vazbami. Detaily viz kapitola 3.3.4.

Důsledky pravidel typicky obsahují příkazy pro modifikaci pracovní paměti (viz kapitola 3.3.3.), tedy přidání (`assert`), odebrání (`retract`), či úpravu (`modify`) faktů v ní. Nemusí tomu tak ale být vždycky - důsledkem aktivace pravidla může být např. vypsání výstupu, logování, zápis souboru, ale také např. ovládání externího systému.

Ukázkový příklad definuje tři odvozovací pravidla. Pravidlo `move-robot` je aktivováno, pokud chceme přesunout nějaký objekt z pozice `?from` na pozici `?to`, objekt se nachází v pozici `?from` a robot nikoli (třetí podmínka je negovaná, viz kapitola 3.3.4.). Poslední podmínka slouží pouze k navázání původní pozice robota. Při aktivaci pravidla je v pracovní paměti nahrazena informace o původní pozici robota pozicí `?from`. Robot se tedy nyní nachází na stejné pozici, jako kýžený objekt.

Podmínky pravidla `move-object` vyžadují, aby byl jak robot, tak objekt určený k přesunu, na pozici `?from`. Při jeho aktivaci je robot i s objektem přesunut na pozici `?to` nahrazením faktů o původních pozicích novými, podobně jako v prvním pravidle. Definice pravidla obsahuje speciální notaci (s použitím operátoru `<-`), jejímž účelem je navázání celého faktu na proměnnou. Ten pak můžeme

v důsledcích snadno ostranit z pracovní paměti. Detaily opět viz kapitola 3.3.4.

Poslední pravidlo slouží k zastavení inference, pokud se již objekt nachází na cílové pozici. Inference je zde zastavena explicitním voláním `(halt)`. Druhou možností by bylo odstranit z pracovní paměti fakt definující cíl (jak vidíme v příkladu 2), neboť v takovou chvíli nemůže být žádné další pravidlo splněno.

Jakmile je znalostní báze nadefinována, můžeme z ní inicializovat pracovní paměť. To provedeme voláním `(reset)`, které, po případném vymazání původních faktů, přidá do pracovní paměti fakty ve všech definovaných skupinách.

Poslední nutnou fází exilového programu je spuštění inference. To můžeme udělat nejjednodušeji voláním `(run)`. Inferenční mechanismus poté postupně vyhodnocuje, která odvozovací pravidla mají splněné všechny podmínky, v každém kroku z nich jedno vybere a aktivuje jej. Detaily viz kapitola 3.3.4.

Výstup programu je následující:

```
==> (IN ROBOT B)
==> (IN BOX A)
==> (GOAL MOVE BOX A B)
Firing MOVE-ROBOT
<== (IN ROBOT B)
==> (IN ROBOT A)
Firing MOVE-OBJECT
<== (IN ROBOT A)
<== (IN BOX A)
==> (IN ROBOT B)
==> (IN BOX B)
Firing STOP
Halting
```

Řádky začínající symbolem `==>` označují fakty přidané do pracovní paměti, řádky začínající `<==` fakty z paměti odstraněné. Tento výstup obdržíme pouze pokud zapneme sledování faktů voláním `(watch facts)` (viz kapitola 3.3.5.). První tři fakty přibydou do pracovní paměti při vyhodnocení volání `(reset)`, další pak s postupnou aplikací odvozovacích pravidel. Dotážeme-li se po skončení inference na seznam faktů v pracovní paměti voláním `(facts)`, obdržíme výstup

```
((GOAL MOVE BOX A B) (IN ROBOT B) (IN BOX B)).
```

Robot i krabice jsou tedy na cílové pozici.

Kód exilového programu má deklarativní charakter. Nikde jsme nemuseli specifikovat, jakou posloupností akcí má systém k výsledku dospět. To nás ovšem nezbavuje nutnosti chápat fungování inferenčního mechanismu ExiLu. Nebudeme-li při konstrukci programu opatrní, může výpočet snadno dospět k neočekávaným výsledkům, dostat se do slepé větve, či se zacyklit. Tyto problémy jsou často způsobeny nezamýšlenou interferencí podmínek pravidel s důsledky jiných.

```
;;; definition of knowledge base
;; facts
(deffacts world
  (in box A)
  (in robot B)
  (goal move box A B))

;; inference rules
(defrule move-robot
  "move robot to object's location"
  (goal move ?object ?from ?to)
  (in ?object ?from)
  (- in robot ?from)
  (in robot ?z)
  =>
  (retract (in robot ?z))
  (assert (in robot ?from)))

(defrule move-object
  "move robot and object to desired location"
  (goal move ?object ?from ?to)
  ?rob-pos <- (in robot ?from)
  ?obj-pos <- (in ?object ?from)
  =>
  (retract ?rob-pos)
  (retract ?obj-pos)
  (assert (in robot ?to))
  (assert (in ?object ?to)))

(defrule stop
  "stop if object is in desired location"
  (goal move ?object ?from ?to)
  (in ?object ?to)
  =>
  (halt))

;;; initialization of working memory
(reset)

;;; inference execution
(run)
```

Příklad 1: Základní struktura exilového programu

Skupina pravidel může například snadno cyklit, pokud první pravidlo závisí na neexistenci nějakého faktu, tento v důsledcích přidá, načež je tento fakt posledním pravidlem odstraněn, aniž by byla v průběhu zneplatněna nějaká další podmínka prvního pravidla. Fakt, který odvodíme v jedné část programu může také neočekávaně splnit podmínku pravidla definovaného v úplně jiné části, která s první neměla vůbec přímo komunikovat.

3.3.2. Definice znalostní báze

ExiL, stejně jako CLIPS, rozlišuje dva typy faktů - jednoduché (*simple, ordered*) a strukturované (*templated*). Stuktura jednoduchého faktu je udána pouze pořadím atomů, typickou volbou je např. `objekt-attribut-hodnota`:

```
(box color red),
```

či `relace-objekty`:

```
(in box hall).
```

Strukturované fakty mají naproti tomu explicitně pojmenované složky (sloty). Typicky popisují objekt s pojmenovanými atributy:

```
(box :color red :size small),
```

či relaci s pojmenovanými zůčastněnými objekty:

```
(in :object box :location hall),
```

kde `box` a `in` jsou šablony (*template*), které je třeba definovat předem. Na pořadí specifikace slotů u strukturovaných faktů nezáleží.

Vyjadřovací síla obou typů faktů je stejná, použitím explicitnějších strukturovaných faktů ale docílíme lepší čitelnosti a jednoznačnější sémantiky exilového programu, zvláště třeba v případě relací na jedné množině objektů:

```
(father john george).
```

Šablonu definujeme voláním makra `deftemplate`, např:

```
(deftemplate in object (location :default here)).
```

Prvním parametrem je název šablony, za ním následuje libovolný počet specifikací slotů. Specifikací slotu je buď symbol - jméno slotu, nebo *seznam*, jehož hlavou (*car*) je jméno slotu a tělem (*cdr*) je *property list (plist)* s dalšími parametry. Aktuálně systém umožňuje pouze specifikaci výchozí hodnoty slotu *klíčkem* `:default`. Ta je použita, není-li při specifikaci faktu, používajícího tuto šablonu, uvedena hodnota pro daný slot. Příklad 2 na straně 24 ukazuje definici znalostní báze ekvivalentní příkladu 1 s použitím strukturovaných faktů.

Je-li už šablona požadovaného názvu definována, ale neexistují v pracovní paměti fakty, které ji používají, je její stávající definice nahrazena. Pokud ale v pracovní paměti nebo v některé z definic skupin faktů existují takové fakty, skončí volání `deftemplate` výjimkou.

Seznam názvů všech definovaných šablon můžeme získat voláním `(templates)`. Specifikaci šablony pak získáme voláním makra `find-template`, např. `(find-template in)`. Definici šablony zrušíme voláním makra `undeftemplate`, např. `(undeftemplate goal)`. To opět skončí výjimkou, existují-li v pracovní paměti nebo v některé ze skupin fakty, které šablonu využívají.

Fakty, ze kterých expertní systém vychází, zavádíme pomocí skupin faktů. Ty definujeme makrem `deffacts`, např.:

```
(deffacts initial
  (goal move box A B)
  (in :object box :location A))
```

Prvním parametrem je název skupiny, pak následuje libovolný počet specifikací faktů. Opakovaným voláním makra `deffacts` je skupina faktů redefinována.

Specifikace faktu je vždy tvořena seznamem. Pokud jde o jednoduchý fakt, specifikací je prostě seznam atomů. Jde-li o fakt strukturovaný, je prvním prvkem specifikace název šablony, za ním následuje plist určující hodnoty slotů faktu. Pokud není hodnota některého slotu uvedena, je buď použita výchozí hodnota, pokud byla v šabloně specifikována, nebo hodnota `nil` v opačném případě.

Seznam názvů všech definovaných skupin faktů získáme voláním `(fact-groups)`. Specifikaci skupiny pak voláním makra `find-fact-group`, např. `(find-fact-group initial)`. Ke zrušení definice skupiny slouží makro `undeffacts` (voláme s názvem skupiny).

Pravidla, pomocí nichž expertní systém během inference odvozuje nové fakty, definujeme makrem `defrule`, např.:

```
(defrule move-robot
  "move robot to object's location"
  (goal :action move :object ?obj :from ?from)
  (in :object ?obj :location ?from)
  (- in :object robot :location ?from)
  ?robot <- (in :object robot :location ?)
  =>
  (modify ?robot :location ?from)).
```

Podmínková část pravidla (před symbolem `=>`) je tvořena vzory. Ty mohou být, stejně jako fakty, jednoduché nebo strukturované. Kromě toho umožňuje definice pravidla několik speciálních konstruktů (negace podmínky, navázání proměnné na celou podmínku). Ty popíšu podrobně v kapitole 3.3.4. spolu s tím, jak jsou podmínky pravidla při inferenci vyhodnocovány.

```
(deftemplate goal (action :default move) object from to)
(deftemplate in object location)

(deffacts world
  (in :object robot :location A)
  (in :object box :location B)
  (goal :object box :from B :to A))

(defrule move-robot
  (goal :object ?obj :from ?from)
  (in :object ?obj :location ?from)
  (- in :object robot :location ?from)
  ?robot <- (in :object robot :location ?)
  =>
  (modify ?robot :location ?from))

(defrule move-object
  (goal :object ?obj :from ?from :to ?to)
  ?object <- (in :object ?obj :location ?from)
  ?robot <- (in :object robot :location ?from)
  =>
  (modify ?robot :location ?to)
  (modify ?object :location ?to))

(defrule stop
  ?goal <- (goal :object ?obj :to ?to)
  (in :object ?obj :location ?to)
  =>
  (retract ?goal))
```

Příklad 2: Definice znalostní báze s použitím strukturovaných faktů

Důsledkovou část pravidla (za symbolem `=>`) tvoří libovolný počet lispových výrazů. Jak se tyto vyhodnocují popíšu opět v kapitole 3.3.4.

Opakovaným voláním makra `defrule` odvozovací pravidlo redefinujeme. K získání seznamu názvů definovaných pravidel a jejich specifikací slouží funkce `rules` a makro `find-rule`, podobně jako u šablon a skupin faktů. Ke zrušení definice pravidla slouží makro `undefrule`.

3.3.3. Modifikace pracovní paměti

Pracovní paměť je množina faktů, které systém v danou chvíli považuje za platné. Její obsah můžeme vypsat voláním funkce `facts`. Funkcí `reset` inicializujeme pracovní paměť ze znalostní báze. Jejím voláním jsou do pracovní paměti zavedeny fakty všech definovaných skupin faktů (viz kapitola 3.3.2.).

Obsah pracovní paměťi může být dále modifikován třemi makry:

- `assert` přidává fakt(y) do pracovní paměti,
- `retract` fakt(y) z pracovní paměti odebírá a
- `modify` přímo modifikuje existující fakt.

Ta lze volat buď před započetím inference (ale po volání `reset`, neboť to pracovní paměť vymaže), nebo v jejím průběhu, pokud inferenci krokujeme (viz kapitola 3.3.4.). Makra také typicky voláme v důsledcích pravidel.

Makra `assert` a `retract` berou jako parametry libovolný počet specifikací faktů ve stejném formátu, jako u makra `deffacts` (ale bez názvu skupiny). Makro `modify` lze použít jen u strukturovaných faktů. Toto makro bere jako první parametr specifikaci faktu, zbytek parametrů tvoří plist určující hodnoty slotů ke změně. Například volání

```
(modify (in :object box :location A) :location B)
```

nahradí v pracovní paměti fakt `(in :object box :location A)` faktem `(in :object box :location B)`. Toto makro je obzvláště užitečné, navážeme-li v podmínkách pravidla celý fakt na proměnnou (viz kapitola 3.3.4.).

Pracovní paměť se skutečně chová jako množina, každý fakt tedy může být v pracovní paměti jen jednou. Opětovné volání `assert` nemá žádný efekt. Volání modify, jehož výsledkem by bylo nahrazení nějakého faktu faktem, který již v pracovní paměti existuje, sice odebere původní fakt, nový však znovu nepřidá.

Všechny fakty můžeme z pracovní paměti odebrat voláním `(retract-all)`. To je ale zřídka užitečné, typicky použijeme spíše funkci `reset` pro navrácení pracovní paměti do výchozího stavu, tedy reinicializaci ze znalostní báze.

3.3.4. Inference

Inference (odvozování nových faktů z aktuálních) probíhá v krocích. V každém kroku jsou vyhodnoceny podmínky všech pravidel, načež je ze splněných pravidel vybráno jedno, které je posléze aktivováno. Inferenční kroky můžeme buď spouštět jednotlivě voláním `(step)`, nebo voláním `(run)` spustit cyklus, který provádí inferenční kroky, dokud je to možné. Cyklus je buď přerušen ve chvíli, kdy není splněno žádné další pravidlo, nebo voláním `(halt)` v důsledcích právě aktivovaného pravidla.

Podmínky pravidel jsou ve tvaru vzorů a jsou spojeny logickou konjunkcí, pravidlo je tedy splněno, jsou-li splněny všechny jeho podmínky. Kromě toho mohou být některé podmínky negovány. Taková podmínka je splněna tehdy, neexistuje-li v pracovní paměti žádný fakt kongruentní s jejím vzorem (při zachování konzistence vazeb proměnných). Negovanou podmínku značí znak - (minus) na prvním místě specifikace vzoru.

Vyhodnocování podmínek pravidel probíhá ve dvou fázích. V první fázi srovnáváme vzory jednotlivých podmínek se všemi fakty v pracovní paměti a to pouze strukturálně, tedy bez ohledu na vazby proměnných. Prvním požadavkem shody je u jednoduchých faktů stejná délka (počet atomů), u strukturovaných faktů stejná šablona. Jednoduchý fakt se nikdy nemůže shodovat se strukturovaným vzorem a naopak.

Dále jsou pak porovnávány jednotlivé atomy (u jednoduchých) či sloty (u složených) faktu vůči odpovídajícímu atomu (slotu) vzoru. Není-li atom (slot) vzoru proměnná, je jednoduše porovnán s atomem faktu. Je-li atomem proměnná, považujeme jej v této fázi automaticky za shodu. Například vzor

```
(in :object robot :location ?loc)
```

se shoduje s faktem

```
(in :object robot :location A)
```

nikoli však s fakty

```
(in :object box :location A)
(is-in :object robot :location A)
(in robot A).
```

Tímto předvýběrem tedy získáme ke každé podmínce pravidla množinu faktů, které mají stejnou strukturu a stejné hodnoty neproměnných atomů.

Ve druhé fázi vyhodnocování hledáme z předvybraných faktů takovou posloupnost (délka odpovídá počtu podmínek pravidla), kde po spárování s odpovídajícími vzory podmínek obdržíme konzistentní vazby proměnných. To znamená, že vyskytuje-li se v podmínkách pravila některá proměnná vícekrát, musí mít odpovídající fakty na daných pozicích stejný atom. Mějme například pravidlo s podmínkami

```
(goal move ?obj ?from ?to)
(in :object ?obj  :location ?from)
(in :object robot :location ?to).
```

Vzor první podmínky je jednoduchý, zatímco další dva jsou strukturované. To ale ničemu nevadí, je třeba pouze najít fakty odpovídající struktury. Posloupnost faktů

```
(goal move box A B)
(in :object box   :location B)
(in :object robot :location A)
```

neprojde druhou fází výběru, neboť vazby proměnných nejsou konzistentní. Proměnná `?from` je například v první podmínce navázána na symbol `A`, v druhé ale na `B`. Kdyby si ovšem krabice s robotem vyměnili pozice, budou vazby proměnných konzistentní a podmínky pravidla budou splněny. Proměnná `?from` by pak měla vazbu `A`, proměnná `?to` vazbu `B` a proměnná `?obj` vazbu `box`.

Vyhodnocení negovaných podmínek si můžeme představit tak, že nejprve vyhodnotíme a navážeme proměnné všech ostatních podmínek. Pokud poté neexistuje fakt kongruentní se vzorem negované podmínky s konzistentními vazbami se zbytkem navázaných proměnných, je tato podmínka splněna. Mějme například pravidlo s podmínkami

```
(goal move box ?from ?to)
(in box ?from)
(- in robot ?from).
```

Máme-li v pracovní paměti pouze fakty

```
(goal move box A B)
(in box A)
(in robot B),
```

budou podmínky pravidla splněny, neboť po spárování vzorů prvních dvou podmínek s prvními dvěma fakty bude proměnná `?from` navázána na hodnotu `A` a neexistuje fakt kogruentní se vzorem `(in robot A)`. Přesuneme-li ale robota na pozici `A`, podmínka již splněna nebude a pravidlo nelze aktivovat. U negovaných podmínek je třeba dát pozor na to, aby se všechny jejich proměnné vyskytovaly už dříve v nějaké pozitivní podmínce. V opačném případě nebude vyhodnocení vazeb proměnných fungovat správně (viz kapitola 3.4.2.).

Ve vzorech podmínek pravidla můžeme využít speciální proměnné `?`. Konzistence vazby této proměnné není při vyhodnocování testována, takže vyskytuje-li se tato proměnná na více místech, chová se tak, jako kdyby byl každý výskyt označen unikátním názvem (podobně jako proměnná _ v Prologu). Použitím této proměnné dáváme najevo, že nás konkrétní hodnota daného atomu nazajímá.

Ve strukturovaných vzorech podmínek není třeba tyto sloty uvádět, neboť `?` je výchozí hodnotou slotu vzoru.

Posledním speciálním konstruktem je navázání celého faktu na proměnnou. Například pravidlo

```
(defrule move
  ?fact <- (in :object ? :location A)
  =>
  (modify ?fact :location B))
```

přesune každý objekt z pozice `A` na pozici `B`. Na proměnnou můžeme navázat i jednoduchý fakt, pak ale nemůžeme použít makra `modify`. Můžeme ovšem volat `(retract ?fact)`, neboť proměnná `?fact` je při aktivaci pravidla nahrazena specifikací faktu, který byl se vzorem podmínky spárován.

Podmínky některých pravidel mohou být při vyhodnocování splněny několika různými posloupnostmi faktů. Výsledkem vyhodnocování pravidel tedy není pouze množina splněných pravidel, nýbrž množina shod (*match*). Každá shoda je tvořena pravidlem a substitucí proměnných navázaných při vyhodnocování jeho podmínek. Substituci chápejme jako zobrazení z množiny všech proměnných, vyskytujících se v podmínkách pravidla, na konkrétní hodnoty atomů (slotů). Aplikací této substituce na vzory podmínek pravidla získáme opět posloupnost faktů, kterými byly podmínky v dané shodě splněny. Tato substituce je následně použita při aktivaci pravidla k náhradě proměnných v jeho důsledcích.

ExiL uchovává aktuální množinu shod. Ta ve skutečnosti není přepočítána v první fázi inferenčního kroku, jak jsem dosud pro jednoduchost tvrdil, nýbrž automaticky po každé změně pracovní paměti či množiny pravidel (detaily viz kapitola 3.4.2.). Aktuální množinu shod nazývám, po vzoru CLIPSu, *agenda* a lze ji vypsat voláním stejnojmenné funkce. Každá shoda v agendě je opatřena časovým razítkem, shody je tedy možné uspořádat podle toho, kdy do agendy přibyly.

Je-li na začátku inferenčního kroku v agendě více shod, je třeba z nich jednu vybrat k aktivaci. Výběr shody záleží na zvolené strategii. ExiL poskytuje následující strategie výběru shody:

depth-strategy	vybírá shodu, která do agendy přibyla nejpozději
breadth-strategy	vybírá shodu, která do agendy přibyla jako první
simplicity-strategy	vybere shodu, jejíž pravidlo má nejméně podmínek
complexity-strategy	volí shodu, jejíž pravidlo má nejvíce podmínek.

Názvy prvních dvou strategií vychází z toho, že jde o prohledávání stromu stavů systému do hloubky, či do šířky (viz kapitola 2.4.).

Výchozí strategií je `depth-strategy`. Strategii, která bude v inferenci použita, můžeme zvolit voláním makra `setstrategy` s názvem strategie,

např. `(setstrategy breadth-strategy)`. Seznam názvů strategií můžeme vypsat voláním `(strategies)`, název aktuálně zvolené strategie pak voláním `(current-strategy)`.

Po výběru shody k aktivaci je její pravidlo aktivováno. Nejprve jsou v důsledcích pravidla nahrazeny všechny proměnné pomocí substituce shody. Výrazy v důsledcích jsou poté vyhodnoceny lispovým makrem `eval`. Tento způsob vyhodnocení vede ke značným omezením. Výrazy totiž nejsou vyhodnoceny v *lexikálním prostředí* definice pravidla, nýbrž ve výchozím (*top-level*) prostředí Lispu. Díky tomu nemůžeme v důsledcích pravidla volat *lokální funkce* či přistupovat k *lokálním proměnným*.

V důsledcích každého pravidla chceme typicky alespoň jednou volat některé z maker modifikujících pracovní paměť, abychom zneplatnili podmínky pravidla, jinak se inference zacyklí. Druhou možností je přerušit inferenci voláním `(halt)`. Volání `(step)` nebo `(run)` ve chvíli, kdy již nelze dále odvozovat (žádné z pravidel nemá splněné podmínky), nemá žádný efekt.

3.3.5. Sledování průběhu inference

ExiL umožnuje sledovat několik typů událostí, ke kterým dochází během inference. K nastavení sledovaných událostí slouží makra `watch` a `unwatch`. K zjištění stavu sledování pak makro `watchedp`.

Základním výstupem programu 1 na straně 21 je

```
Firing MOVE
Firing PUSH
Firing STOP
Halting.
```

Zapneme-li sledování faktů voláním `(watch facts)`, obdržíme výstup

```
==> (IN BOX A)
==> (IN ROBOT B)
==> (GOAL MOVE BOX A B)
Firing MOVE-ROBOT
<== (IN ROBOT B)
==> (IN ROBOT A)
Firing MOVE-OBJECT
<== (IN ROBOT A)
<== (IN BOX A)
==> (IN ROBOT B)
==> (IN BOX B)
Firing STOP
Halting.
```

Sledování pravidel - `(watch rules)` - přidává informace o pravidlech přidaných do (odebraných ze) znalostní báze, například

```
==> (RULE STOP
  (GOAL MOVE ?OBJECT ?FROM ?TO)
  (IN ?OBJECT ?TO)
  =>
  (HALT)).
```

Po zapnutí sledování agendy (voláním `(watch activations)`, název je kvůli kompatibilitě se systémem CLIPS) budeme navíc informováni o shodách, které do agendy přibyly, nebo z ní byly odstraněny. Výstup programu pak bude následující:

```
==> (MATCH MOVE-ROBOT
      ((GOAL MOVE BOX A B) (IN BOX A) NIL (IN ROBOT B)))
Firing MOVE-ROBOT
==> (MATCH MOVE-OBJECT
      ((GOAL MOVE BOX A B) (IN BOX A) (IN ROBOT A)))
==> (MATCH MOVE-ROBOT
      ((GOAL MOVE BOX A B) (IN BOX A) NIL (IN ROBOT A)))
<== (MATCH MOVE-ROBOT
      ((GOAL MOVE BOX A B) (IN BOX A) NIL (IN ROBOT A)))
Firing MOVE-OBJECT
==> (MATCH STOP
      ((GOAL MOVE BOX A B) (IN BOX B)))
Firing STOP
Halting
```

Každá shoda je zde reprezentována názvem pravidla a posloupností faktů, které byly spárovány s jeho podmínkami. Odtud můžeme snadno odvodit substituci, jež byla při vyhodnocení použita. Negované podmínky nejsou spárovány s žádným konkrétním faktem, proto jsou na odpovídajících pozicích hodnoty `NIL`.

Je zde také vidět, že po aktivaci pravidla `move-robot` se v agendě na chvíli objeví opětovná shoda tohoto pravidla. To je způsobeno tím, že obsah agendy se přepočítává po každé změně pracovní paměti, takže se zde mohou objevit dočasné výsledky.

3.3.6. Undo/redo

Jedním z implementovaných rozšíření původního programu je schopnost vrácení provedených změn. K tomu slouží makra `undo` a `redo`. Ta lze použít k vrácení jakékoli akce s vedlejším efektem, včetně kroků inference. K vypsání zásobníků s akcemi, které je možné vrátit, jsou k dispozici makra `undo-stack` a `redo-stack`.

Pokud například vyhodnotíme prvních 32 řádků programu 1 na straně 21 a zavoláme dvakrát `undo`, bude výpis zásobníků následující (přeformátováno):

```
EXIL-USER> (undo-stack)
  1: (defrule MOVE-ROBOT
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        (IN ?OBJECT ?FROM)
        (- IN ROBOT ?FROM) (IN ROBOT ?Z)
        =>
        (RETRACT (IN ROBOT ?Z))
        (ASSERT (IN ROBOT ?FROM))))
  2: (deffacts WORLD
       ((IN BOX A) (IN ROBOT B) (GOAL MOVE BOX A B)))

EXIL-USER> (redo-stack)
  1: (defrule MOVE-OBJECT
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        ?OBJ-POS <- (IN ?OBJECT ?FROM)
        ?ROB-POS <- (IN ROBOT ?FROM)
        =>
        (RETRACT ?ROB-POS)
        (RETRACT ?OBJ-POS)
        (ASSERT (IN ROBOT ?TO))
        (ASSERT (IN ?OBJECT ?TO))))
  2: (defrule STOP
       ((GOAL MOVE ?OBJECT ?FROM ?TO)
        (IN ?OBJECT ?TO)
        =>
        (HALT)))
```

Vidíme tedy, že jsme vrátili zpět definice pravidel `move-object` a `stop`, ty bychom mohli opět provést voláním `(redo)`. Dalším voláním `(undo)` by pak byla vrácena definice pravidla `move-robot` a poté definice skupiny faktů `world`.

Nemá-li akce žádný vedlejší efekt - např. volání `assert` s faktem, který už v pracovní paměti je, či volání `run` ve chvíli, kdy už není co odvozovat - prázdná akce se na zásobník neuloží. ExiL umožňuje vrácení akcí až 20 kroků zpět.

3.3.7. Zpětná inference

Dalším z implementovaných rozšíření je možnost zpětné inference. Inference popsaná v kapitole 3.3.4. je dopředná. V každém kroku jsou nalezeny všechny možnosti dalšího postupu odvozování, načež je zvolena jedna, kterou se program dále ubírá. To činí průběh inference značně nedeterministickým. Možnosti postupu, které nebyly vybrány, mohou být navíc dalším postupem ztraceny, pokud

aktivace některého pravidla zneplatní podmínky jiného. Míru nedeterminismu můžeme snížit tím, že budeme navrhovat odvozovací pravidla tak, aby se výpočet neubíral nechtěnými cestami. To ale není vždy jednoduché, nebo dokonce možné.

Zpětná inference naproti tomu umožňuje definovat cíle, kterých chceme dosáhnout. K tomu slouží makro `defgoal`, kterému předáme vzor ve stejném formátu, jako u podmínek pravidel. Definice cíle ovšem nepodporuje negaci ani navázání faktu na proměnnou (k tomu ani není důvod). Cíle je pak možné vypsat voláním `(goals)`. Cíl můžeme také odebrat makrem `undefgoal`. Ke spuštění zpětné inference slouží funkce `back-step` a `back-run`, podobně jako u inference dopředné.

Mějme následující znalostní bázi:

```
(deffacts world
  (have-money))

(defrule buy-car
  (have-money)
  =>
  (retract (have-money))
  (assert (have-car)))

(defrule pay-rent
  (have-money)
  =>
  (retract (have-money))
  (assert (rent-payed))).
```

Spustíme-li dopřednou inferenci, systém nám vesele doporučí nákup auta. Mít nové auto je sice pěkné, hrozí-li nám ale vykázání z pronajatého bytu, není nákup auta pravděpodobně cestou, kterou bychom se chtěli ubírat. Systém by nám v tuto chvíli mohl stejně dobře doporučit správnou cestu. Že bylo vybráno zrovna první pravidlo je výsledkem toho, jak funguje síť RETE, která pravidla vyhodnocuje. Za daných okolností ale nechceme špatnou variantu ani připouštět.

V tomto případě bychom mohli upravit definici programu tak, že do znalostní báze přidáme informaci o cíli, kterou budou pravidla zohledňovat, podobně jako v příkladu 1 na straně 21. Muset ale programovat zohlednění cíle v každém pravidle je přinejmenším otravné. U větších programů to navíc může být velmi náročné, neboť cíl bude třeba programově modifikovat v průběhu výpočtu.

S použitím zpětné inference je problém podstatně jednodušší. Zavoláme-li

```
(reset)
(defgoal (rent-payed))
(back-run),
```

bude výsledkem výstup

```
All goals have been satisfied
(RENT-PAYED) satisfied by (RULE PAY-RENT
  (HAVE-MONEY)
  =>
  (RETRACT (HAVE-MONEY))
  (ASSERT (RENT-PAYED)))
(HAVE-MONEY) satisfied by (HAVE-MONEY).
```

Zde vidíme, že po spuštění zpětné inference nezačal systém bezhlavě provádět akce, ke kterým měl dostatečné prostředky. Místo toho systém uvážil zadaný cíl a jal se hledat akce, které k vedou jeho splnění.

Uvažme nyní složitější příklad (definice šablon vynechána):

```
(deffacts world
  (female jane)
  (male john)
  (parent :parent jane :child george)
  (parent :parent john :child george))

(defrule father-is-male-parent
  (male ?father)
  (parent :parent ?father :child ?child)
  =>
  (assert (father :father ?father :child ?child)))

(defrule mother-is-female-parent
  (female ?mother)
  (parent :parent ?mother :child ?child)
  =>
  (assert (mother :mother ?mother :child ?child))).
```

Zajímá-li nás, kdo je matkou George, můžeme zkusit spustit dopřednou inferenci. Po jejím skončení bude v pracovní paměti jak informace o Georgově matce, tak o jeho otci. Systém se tedy v tomto případě dobral správného výsledku, vypočítal ale i další fakty, které nás nezajímaly. Dokážeme si snadno představit, že ve větším programu může být výpočet všech odvoditelných závěrů velmi výpočetně a tudíž i časově náročný.

Spustíme-li naopak zpětnou inferenci voláním

```
(reset)
(defgoal (mother :mother ?mother-of-george :child george))
(back-run),
```

je výsledkem

```
All goals have been satisfied
(MOTHER (MOTHER . ?MOTHER-OF-GEORGE) (CHILD . GEORGE)) satisfied by
  (RULE MOTHER-IS-FEMALE-PARENT
    (FEMALE ?MOTHER)
    (PARENT (PARENT . ?MOTHER) (CHILD . ?CHILD))
    =>
    (ASSERT (MOTHER :MOTHER ?MOTHER :CHILD ?CHILD)))
(FEMALE ?MOTHER) satisfied by (FEMALE JANE)
(PARENT (PARENT . JANE) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
These variable bindings have been used:
((?MOTHER-OF-GEORGE . JANE)).
```

Systém tedy vyvodil pouze závěr, který nás zajímal.

Zpětná inference umožnuje také výpočet alternativních odpovědí, tedy hledání dalších cest výpočtu (a vazeb proměnných), které vedou ke splnění všech cílů. Na další alternativní odpověď se dotážeme jednoduše opětovným voláním `(back-run)`.

Zajímají-li nás například oba rodiče George, můžeme zadat cíl

```
(defgoal (parent :parent ?parent :child george)).
```

Pokud poté třikrát zavoláme `(back-run)`, obdržíme výstup

```
All goals have been satisfied
(PARENT (PARENT . ?PARENT) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
These variable bindings have been used:
((?PARENT . JANE))

All goals have been satisfied
(PARENT (PARENT . ?PARENT) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JOHN) (CHILD . GEORGE))
These variable bindings have been used:
((?PARENT . JOHN))

No feasible answer found.
```

Systém tedy najde obě možné odpovědi (vazby proměnných), vedoucí ke splnění cíle, načež oznámí, že další odpověď už neexistuje.

Zpětná inference nemění obsah pracovní paměti. To ani není možné vzhledem k tomu, že ve chvíli, kdy inference zvolí pravidlo, jehož důsledky vedou ke splnění aktuálního cíle, nejsou jeho podmínky často ještě splněny. Místo toho pracuje zpětná inference pouze s množinou cílů.

Jednotlivé cíle jsou postupně vybírány a hledají se cesty k jejich splnění. Inference nejprve zkoumá fakty v pracovní paměti. Není-li aktuální cíl splněn žádným z platných faktů, uvažuje dále jednotlivá pravidla. Najde-li pravidlo, které by po aktivaci vedlo ke splnění aktuálního cíle, naváže proměnné, použité v jeho důsledcích, podle vzoru cíle. Tyto vazby poté aplikuje na jeho podmínky (tedy opačně než u inference dopředné) a ty pak přidá do množiny cílů. Použité vazby proměnných se navíc aplikují i na zbytek cílů.

Zkusme nyní krokovat (použitím `back-step`) předchozí příklad s dotazem na matku George s průběžným výpisem cílů pomocí `(goals)`.

```
GOALS: ((MOTHER :MOTHER ?MOTHER-OF-GEORGE :CHILD GEORGE))
(MOTHER (MOTHER . ?MOTHER-OF-GEORGE) (CHILD . GEORGE)) satisfied by
  (RULE MOTHER-IS-FEMALE-PARENT
    (FEMALE ?MOTHER)
    (PARENT (PARENT . ?MOTHER) (CHILD . ?CHILD))
    =>
    (ASSERT (MOTHER :MOTHER ?MOTHER :CHILD ?CHILD)))

GOALS: ((FEMALE ?MOTHER) (PARENT :PARENT ?MOTHER :CHILD GEORGE))
(FEMALE ?MOTHER) satisfied by (FEMALE JANE)

GOALS: ((PARENT :PARENT JANE :CHILD GEORGE))
(PARENT (PARENT . JANE) (CHILD . GEORGE)) satisfied by
  (PARENT (PARENT . JANE) (CHILD . GEORGE))
```

V prvním kroku je proměnná `?mother-of-george`, použitá v definici cíle, nahrazena proměnnou `?mother`, použitou v důslecích pravidla `mother-if-female-parent`. Proměnná `?child` v důsledcích je navázána na `george` a touto vazbou jsou nahrazeny výskyty proměnné v podmínkách pravidla. Podmínky jsou poté přidány do množiny cílů. Původní cíl je poté z množiny odstraněn.

V druhém kroku je nový cíl `(female ?mother)` porovnán s faktem `(female jane)` v pracovní paměti, je jím splněn a v posledním cíli je proměnná `?mother` navázána na `jane`. Tento cíl je pak v posledním kroku triviálně splněn identickým faktem.

Aktuální stav výpočtu pomocí zpětné inference je ztracen, zavoláme-li během krokování `defgoal` nebo `undefgoal`. Kdyby tomu tak nebylo, mohli bychom systém uvést do nekonzistentního stavu.

Implementace zpětné inference v ExiLu je velmi omezená. V důsledcích pravidel zohledňuje pouze specifikace faktů ve voláních `assert`. Nelze ji tedy aplikovat u pravidel, která fakty z pracovní paměti odstraňují, či je modifikují. Inference také neumí pracovat s negovanými cíli, tudíž ani s pravidly, která mají negované podmínky (neboť tyto by při použití pravidla mezi cíli objevily).

Použití zpětné inference nemusí nutně snížit míru nedeterminismu výpočtu. Existuje-li několik pravidel, která vedou ke splnění aktuálního cíle, bude výpočet opět nedeterministický. Na rozdíl od dopředné inference lze ale alternativní cesty výpočtu postupně prohledat. To je možné jednak díky backtrackingu - nevede-li daná cesta ke splnění všech cílů, výpočet se vrátí a zkusí se ubírat jinudy. Dále díky možnosti dotázat se na alternativní odpovědi.

3.3.8. Reset prostředí

Objekt, která uchovává aktuální stav systému, nazývám v ExiLu prostředím. Hodnoty tvořící aktuální stav prostředí rozděluji do dvou skupin. První skupinu tvoří hodnoty trvalé. Sem spadají definice šablon, znalostní báze (skupiny faktů, pravidla), definice strategií pro volbu shod a aktuálně zvolená strategie.

Druhou skupinou jsou hodnoty dočasné. Sem patří aktuální stav pracovní paměti, aktuální cíle, stav sítě RETE spolu s agendou udržující aktuální shody (splněná pravidla spolu s vazbami proměnných), hodnoty zásobníků sloužících k vrácení provedených akcí a jejich opětovné provedení (undo/redo) a hodnota zásobníku udržujícího aktuální stav zpětné inference (aby bylo možné ji krokovat nebo se dotázat na alternativní odpovědi).

Dočasné hodnoty lze uvést do výchozího stavu voláním `(clear)`. Tím systém uvedeme do stavu, kdy zná pouze definované šablony, skupiny faktů a odvozovací pravidla. Volání `(reset)` provede `(clear)` následovaný zavedením všech skupin faktů do pracovní paměti. Všechny hodnoty prostředí, tedy včetně trvalých, lze pak uvést do výchozího stavu voláním `(complete-reset)`.

Chování `reset` a `clear` vzhledem k zásobníkům pro vracení akcí může být poněkud překvapivé. Makro `reset` totiž vymaže obsah tohoto zásobníku, poté na něj však uloží akci k vrácení jeho volání. Z výpisu `(undo-stack)` tak po volání `reset` zmizí předchozí akce, uvidíme zde jen akci `(reset)`. Po volání `(undo)` se na zásobníku předchozí akce opět objeví. Totéž platí pro volání `(clear)`.

3.3.9. Práce s více prostředími

Při práci s ExiLem se nemusíme omezovat pouze na jedno prostředí (byť si s ním často vystačíme). Nové prostředí lze definovat voláním `defenv` s názvem prostředí, např. `(defenv test)`. Prostředí pak lze přepnout voláním `(setenv test)`, případně smazat voláním `(undefenv test)`.

Každé prostředí má oddělený stav, tedy trvalé i dočasné hodnoty (viz předchozí kapitola). Na název aktuálního prostředí se lze dotázat voláním `(current-environment)`. Název výchozího prostředí je `default`. Seznam všech prostředí získáme voláním `(environments)`.

Máme-li už definované prostředí, např. `test`, opětovné volání `(defenv test)` skončí výjimkou. Tím je zajištěno, že si omylem nevymažeme celé prostředí. Chceme-li jej opravdu vymazat, musíme volat `(defenv test :redefine t)`.

3.3.10. Volání ExiLu z jiného kódu a naopak

Doposud jsem v ukázkách práce s ExiLem pracoval většinou s makry. Byla to následující:

definice šablon	`deftemplate undeftemplate find-template`
definice skupin faktů	`deffacts undeffacts find-fact-group`
definice pravidel	`defrule undefrule find-rule`
modifikace pracovní paměti	`assert retract modify`
definice cílů	`defgoal undefgoal`
sledování průběhu inference	`watch unwatch watchedp`
strategie výběru shody	`defstrategy setstrategy`
definice prostředí	`defenv undefenv setenv`

Tato makra berou jako parametry symboly a/nebo seznamy a tyto automaticky *kvotují*. To je pohodlné, pracujeme-li s knihovnou přímo. Představme si ale, že chceme knihovnu volat z jiného kódu a například specifikace faktů, které předáváme makru `assert`, generovat nějakou funkcí.

Protože makro `assert` seznamy se specifikacemi faktů kvotuje, místo aby je vyhodnotilo, nelze ho v tomto případě použít. Výsledkem volání

```
(assert (generate-fact))
```

totiž bude přidání faktu `(generate-fact)` do pracovní paměti. K tomuto účelu poskytuje ExiL ke všem uvedeným makrům funkční alternativy, které parametry vyhodnocují. Tyto jsou označeny suffixem `f`, například `assertf`. Tímto suffixem sice v Lispu typicky označujeme destruktivní makra, která mění svůj argument, v případě exilových maker ale záměna nehrozí.

Dotazovací funkce a makra jako `facts`, `goals`, `find-fact-group`, apod. navíc nevypisují hodnoty na výstup, nýbrž vrací externí reprezentaci objektů, se kterou je možné dále manipulovat a pak ji třeba systému předat zpátky. To umožnuje například následující (nepříliš užitečné) volání:

```
(dolist (fact (facts))
  (retractf fact)
  (assertf (cons 'my-fact fact))).
```

Funkční alternativy jako `assertf` také umožnují použití složitějších konstruktů v důsledcích pravidla. Bude-li například podmínka pravidla

```
(defrule surround-by-as
  (palindrome ?p)
  =>
  (assertf `(palindrome ,(concatenate 'string "a" ?p "a"))))
```

splněna faktem `(palindrome "b")`, přibyde po jeho aktivaci do pracovní paměti fakt `(palindrome "aba")`. Funkci `assertf` bohužel nelze použít se zpětnou inferencí, neboť ta nemá šanci předvídat, jaký fakt bude jejím voláním do pracovní paměti přidán.

V důsledcích pravidla bychom také mohli chtít například upozornit jinou část programu na událost, ke které došlo. Protože ExiL nahrazuje výskyty proměnných v celé důsledkové části pravidla, lze toho snadno dosáhnout. Je ale třeba dát pozor na kvotování hodnot proměnných. Uvažme pravidlo

```
(defrule move-robot
  (goal move robot ?from ?to)
  (in robot ?from)
  =>
  (retract (in robot ?from))
  (assert (in robot ?to))
  (notify 'moving-robot '?from '?to))
```

a fakt `(goal move robot A B)` v pracovní paměti. Kdybychom ve volání `notify` proměnné `?from` a `?to` nekvotovali, volání by se při aktivaci pravidla vyhodnotilo jako `(notify 'moving-robot A B)`. To by pravděpodobně skončilo chybovou hláškou sdělující, že proměnné `A` a `B` nejsou definovány.

Makra `deftemplate`, `deffacts`, `defrule` a `modify` berou specifikace slotů šablony, faktů, těla pravidla a změn k provedení (v tomto pořadí) jako další parametry. Díky tomu nemusíme tyto parametry obalovat do dalšího seznamu. Jejich funkční alternativy naproti tomu očekávají tyto parametry v jednom seznamu, například

```
(deffactsf 'world (list '(in box A) '(in robot B))).
```

To umožňuje snazší generování těchto specifikací funkcemi.

3.3.11. Kompatibilita se systémem CLIPS

Dalším z požadavků zadání práce bylo přiblížit syntaxi exilových volání systému CLIPS, aby bylo možné programy v něm napsané snáze převést na programy exilové. Toho se mi podařilo dosáhnout jen částečně.

```
(deftemplate goal
  (slot action (default move))
  (slot object)
  (slot from)
  (slot to))

(deftemplate in
  (slot object)
  (slot location))

(deffacts world
  (in (object robot) (location A))
  (in (object box) (location B))
  (goal (object box) (from B) (to A))).

(defrule move-robot
  (goal (action move) (object ?obj) (from ?from))
  (in (object ?obj) (location ?from))
  (- in (object robot) (location ?from))
  ?robot <- (in (object robot) (location ?z))
  =>
  (modify ?robot (location ?from)))

(defrule move-object
  (goal (action move) (object ?obj) (from ?from) (to ?to))
  ?object <- (in (object ?obj) (location ?from))
  ?robot <- (in (object robot) (location ?from))
  =>
  (modify ?robot (location ?to))
  (modify ?object (location ?to)))

(defrule stop
  ?goal <- (goal (action move) (object ?obj) (to ?to))
  (in (object ?obj) (location ?to))
  =>
  (halt))
```

Příklad 3: Definice znalostní báze s použitím CLIPSové syntaxe

Systém CLIPS používá jiný formát specifikací slotů šablony, strukturovaných faktů a požadovaných změn při volání `modify`. Tuto syntaxi nyní ExiL podporuje také. Příklad 3 na straně 39 ukazuje definici znalostní báze ekvivalentní příkladu 2 s použitím CLIPSové syntaxe. Syntax je dostatečně odlišná na to, aby ji ExiL rozpoznal, není tedy třeba syntaktický mód nijak přepínat. Díky tomu dokonce můžeme oba typy syntaxe kombinovat.

ExiL také po vzoru CLIPSu umožňuje omezit seznam vrácený voláním `(facts)` volitelnými číselnými parametry. První volitelný parametr udává index prvního faktu v seznamu (číslováno od 1). Druhý parametr udává index posledního faktu. Třetí parametr pak maximální počet vrácených faktů.

Makra `assert` a `retract` také nyní umožňují přidání či odebrání více faktů najednou. Makru `retract` lze navíc místo specifikací faktů k odstranění předat jejich číselné indexy v seznamu faktů. Obě možnosti lze dokonce kombinovat.

V podmínkách pravidel lze nyní také použít speciální proměnnou `?` a navázání celého faktu na proměnnou pomocí operátoru `<-`, jak jsem již popsal v kapitole 3.3.4.

CLIPS dále umožňuje specifikovat u slotu šablony datový typ. To považuji v dynamickém jazyce, jakým je Common Lisp, za zbytečné. CLIPS také nabízí několik vlastních funkcí a možnost definice nových, které pak lze používat v důsledcích pravidel. To nemá v ExiLu smysl, neboť můžeme použít vestavěné nebo uživatelsky definované funkce Lispu.

CLIPS navíc poskytuje možnost definovat některé sloty šablony jako *multislot*. Ve vzorech podmínek pravidel pak můžeme používat proměnnou `$?`, která se naváže na jeden nebo více atomů multislotu, případně jednoduchého faktu. Dále je v podmínkách CLIPSu možné používat libovolně vnořené agregační funkce `and`, `or`, `not` a podobně. Ve vzorech podmínek CLIPSových pravidel lze také používat speciální proměnné, které se shodují jen s některými atomy, např. proměnná `?color&~red&~yellow` se shoduje se všemi atomy, kromě `red` a `yellow`.

Tyto možnosti ExiL neposkytuje, neboť jejich implementace by vyžadovala přepsat velkou část algoritmu RETE, který podmínky pravidel vyhodnocuje, jak popíšu v kapitole 3.4.3..

3.3.12. Grafické uživatelské rozhraní

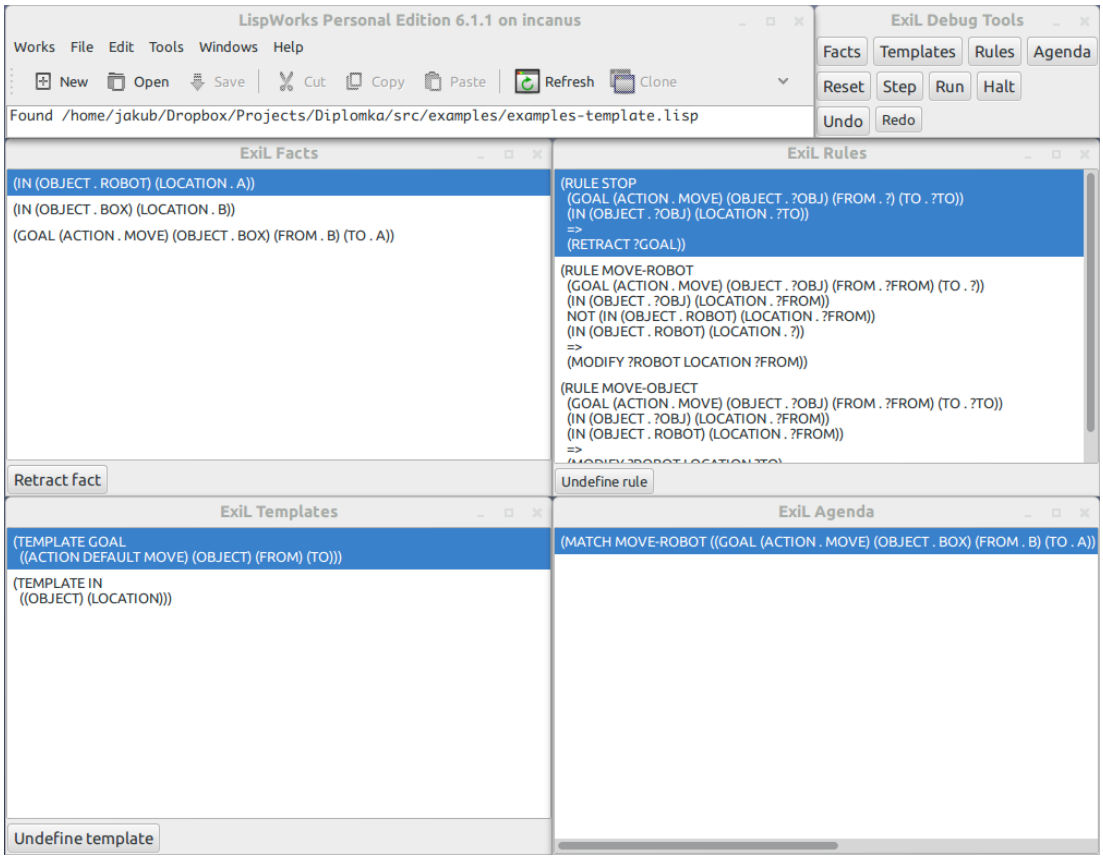
Pro prostředí LispWorks jsem k ExiLu implementoval minimalistické grafické uživatelské rozhraní, zobrazené na obrázku 1. To sestává z hlavního okna s deseti tlačítky organizovanými do tří řad (vpravo nahoře).

První řada tlačítek - „Facts“, „Templates“, „Rules“ a „Agenda“ slouží k zobrazení podoken s jednotlivými položkami. Okno „Facts“ zobrazuje seznam faktů v pracovní paměti a umožňuje jejich odebrání tlačítkem „Retract fact“. Okno „Templates“ zobrazuje seznam definovaných šablon a umožňuje jejich odebrání tlačítkem „Undefine template“. Okno „Rules“ zobrazuje seznam definovaných odvozovacích pravidel a umožňuje jejich odebrání tlačítkem „Undefine rule“. Po-

slední okno „Agenda“ zobrazuje seznam aktuálních shod v agendě. Seznamy v oknech se automaticky obnovují při každé změně zobrazených hodnot.

Druhá řada tlačítek - „Reset“, „Step“, „Run“ a „Halt“ umožňuje řízení inference. Tlačítka volají stejnojmenné funkce ExiLu.

Poslední řada tlačítek - „Undo“ a „Redo“ slouží k vracení a opětovnému provedení akcí voláním stejnojmenných funkcí ExiLu.



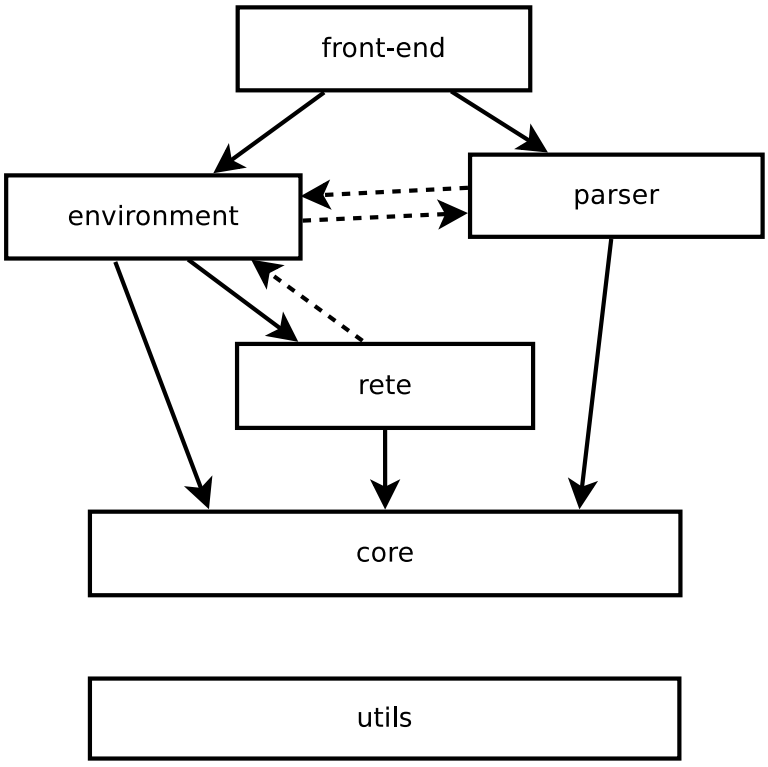
Obrázek 1. Grafické uživatelské rozhraní

Rozhraní lze zobrazit voláním `(exil-gui:show-gui)`. Každé prostředí má vlastní rozhraní. Volání `show-gui` bez parametru zobrazí rozhraní k aktuálnímu prostředí. Jako volitelný parametr můžeme funkci předat název prostředí, jehož rozhraní chceme zobrazit. Máme-li tedy definováno více prostředí, můžeme si ke každému z nich zobrazit uživatelské rozhraní. Po načtení knihovny v prostředí LispWorks se automaticky zobrazí uživatelské rozhraní pro výchozí prostředí.

3.4. Implementace

3.4.1. Architektura programu

V následující kapitole budu používat obecný pojem *modul* pro ohraničenou část programu, implementující nějakou část funkcionality. Veřejné rozhraní modulu definuje vstupní body, pomocí nichž s ním mohou ostatní moduly komunikovat. Ty jsou zde reprezentovány funkcemi, makry a metodami, které modul poskytuje a specifikacemi jejich parametrů. V Common Lispu sice tuto entitu nazýváme *package*, tento pojem se však špatně skloňuje a celkově narušuje plynulost textu.



Obrázek 2. Architektura ExiLu

Obrázek 2. zobrazuje architekturu knihovny ExiL, tedy moduly, do kterých je kód knihovny rozdělen, a jejich vzájemnou komunikaci. Orientace šipek určuje směr volání funkcí (metod, maker) poskytovaných moduly, např. modul `front-end` volá funkce modulu `parser` a ne naopak. Těmito voláními jsou zároveň definovány závislosti mezi moduly, ty mají tedy stejnou orientaci.

Modul `utils` definuje různorodé pomocné funkce a makra, která jsou volána všemi ostatními moduly. V obrázku 2. bych tedy mohl zahrnout šipky ze všech ostatních modulů do modulu `utils`, to by jej však činilo zbytečně nepřehledným. Většina funkcí a maker, která modul definuje, usnadňuje nízkoúrovňovou práci se

symboly a datovými strukturami - (asociativními) seznamy, množinami, stromy, *hashovacími tabulkami* apod.

Modul `core` definuje základní objekty, s nimiž zbytek knihovny pracuje. Ty reprezentují fakty, vzory, šablony a pravidla. Fakty a vzory jsou definovány ve dvou variantách - jednoduché a strukturované, jak jsem uvedl v uživatelské příručce.

Modul `rete` implementuje algoritmus RETE, který slouží k efektivnímu vyhodnocování podmínek odvozovacích pravidel. Algoritmus je implementován *dataflow* sítí, jejíž fungování popíšu detailně v kapitole 3.4.2. Algoritmus RETE je sice nejkomplexnější částí ExiLu, veřejné rozhraní modulu `rete` je však velmi jednoduché.

Modul poskytuje dvojici metod, kterými síť RETE upozorníme na přidání nebo odebrání pravidel. Ty dle potřeby doplní síť o uzly potřebné k vyhodnocení podmínek těchto pravidel, případně uzly při odebrání pravidla odstraní.

Další dvojice metod upozorní síť na přidání faktu do (resp. odebrání z) pracovní paměti. Síť pak přepočítá splněné podmínky pravidel a případně upozorní prostředí na nově přibyvší nebo rozbité shody (*broken match* - pravidla, která byla splněná nějakou posloupností faktů a už nejsou).

Modul `environment` definuje stejnojmenný objekt, který udržuje průběžný stav prostředí a řídí průběh inference. Hodnoty, které stav prostředí tvoří, jsem popsal v kapitole 3.3.8. Prostředí udržuje krom jiného referenci na síť `rete`, kterou upozorňuje na nastalé změny.

Modul `parser` zajišťuje vytváření `core` objektů z externí reprezentace (té, kterou předáváme funkcím a makrům při práci s knihovnou). Modul rozpoznává jak základní, tak CLIPSovou syntax, takže o to, kterou z nich uživatel používá, se zbytek kódu nemusí dále starat.

Při parsování strukturovaných faktů či vzorů se `parser` dotazuje aktuálního prostředí na definici šablony. Při zpětné inferenci prostředí vyhodnocuje volání `assert` v důsledcích pravidel a žádá `parser` o zparsování reprezentací faktů v nich. Proto jsem mezi moduly `environment` a `parser` přidal šrafované šipky. Kromě těchto nutných interakcí spolu však moduly nekomunikují.

Modul `front-end` udržuje seznam definovaných prostředí a ví, které je právě aktivní. Kromě manipulace tohoto seznamu modul sám žádnou funkcionalitu neimplementuje. Pouze kvotuje parametry předané makrům, předává je `parser`u a výsledné objekty (se zbytkem parametrů) pak aktivnímu prostředí.

Do diagramu jsem nezahrnul modul `gui`. Ten implementuje grafické uživatelské rozhraní a poskytuje dvě metody - `show-gui` a `update-lists`. Rozhraní se dotazuje prostředí, se kterým je svázáno, na hodnoty aktuálního stavu a je jím upozorněno, pokud se tyto změní.

3.4.2. Algoritmus RETE

Algoritmus RETE slouží k efektivnímu vyhodnocování splnění podmínek odvozovacích pravidel (dále jen vyhodnocování). Je implementován dataflow sítí. Ta je rozdělena do dvou částí, označovaných jako alpha a beta. Alpha část sítě vyhodnocuje splnění jednotlivých podmínek pravidla fakty nově přidanými do či odebranými z pracovní paměti. Beta část pak zajišťuje zachování konzistence vazeb proměnných mezi podmínkami.

To, že je vyhodnocování sítí RETE tak efektivní, je zajištěno maximálním sdílením částí sítě mezi pravidly. Mají-li dvě pravidla strukturálně stejný vzor nějaké podmínky (liší se jen použité proměnné), sdílí tato pravidla část alpha sítě, která tuto podmínku vyhodnocuje. Pokud mají dvě pravidla stejných několik prvních podmínek, sdílí tato pravidla kromě části alpha sítě i část beta sítě, která zajišťuje konzistenci vazeb proměnných mezi nimi. Čím více je v systému definováno odvozovacích pravidel, tím pravděpodobnější jsou shody mezi jejich podmínkami a tím více se efektivita algoritmu projeví.

Dataflow síť tvoří orientovaný acyklický graf[31]. Uzly sítě rozdělujeme na dva základní typy - paměťové a testovací. Paměťové uzly ukládají průběžné výsledky vyhodnocování. Testovací uzly provádějí různé typy testů (v závislosti na typu uzlu) nutných pro vyhodnocení podmínek pravidla.

Mezi uzly sítě dochází ke dvěma typům interakcí. Hlavním typem interakce je aktivace. Ta probíhá ve směru hran grafu a uzel při ní provádí svou hlavní funkci v závislosti na typu. Testovací uzel se dále může dotazovat sousedního paměťového uzlu na obsah jeho paměti a to **proti směru orientace hran**. Abych odlišil sousední uzly ve směru hran (tedy ty, jež může uzel aktivovat) od množiny všech sousedů, budu první skupinu nazývat *potomky*.

Uzlu je při aktivaci vždy předána nějaká datová struktura. Tou je buď fakt, nebo token, který reprezentuje posloupnost faktů. Zatímco paměťové uzly po aktivaci a uložení datové struktury do své paměti vždy aktivují své potomky, testovací uzly aktivují potomky jen v případě úspěšného testu.

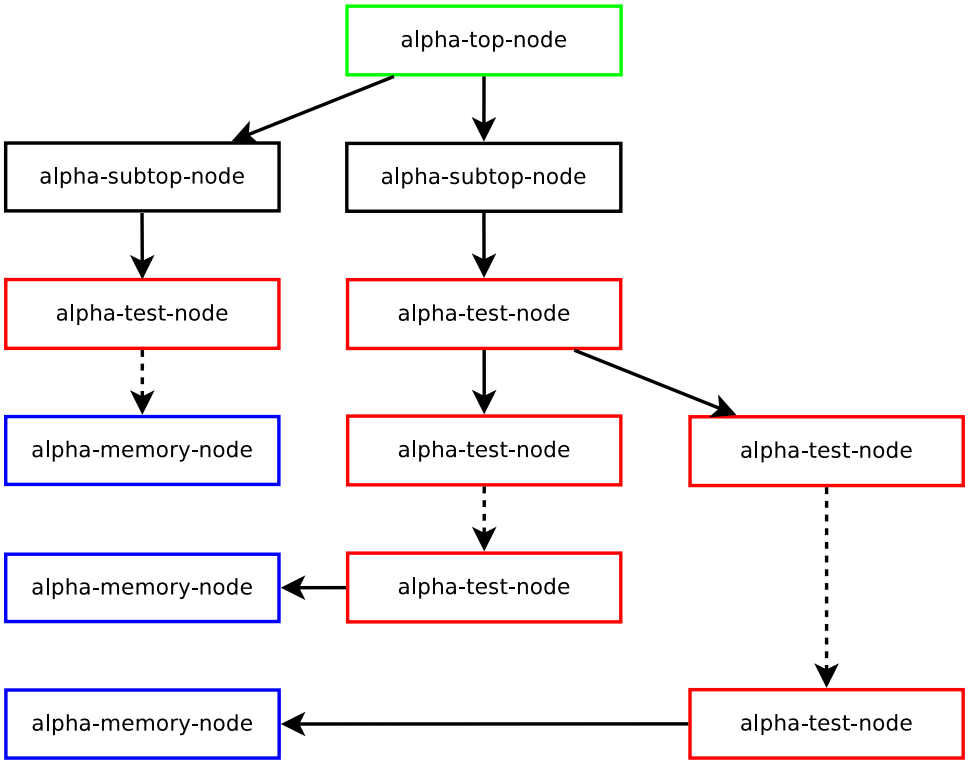
Obrázek 3. zobrazuje úsek alpha části sítě RETE. Uzel `alpha-top-node` je vstupním bodem této sítě. Ten je aktivován při každé změně pracovní paměti faktem, který byl přidán nebo odebrán. Potomky uzlu `alpha-top-node` jsou uzly `alpha-subtop-node`, jeden pro jednoduché fakty a jeden pro každou definovanou šablonu. Uzel `alpha-top-node` aktivuje příslušný `alpha-subtop-node` podle typu faktu. Ten pak aktivuje `alpha-test-node` pod ním.

Šrafované šipky v obrázku 3. symbolizují, že zde může následovat posloupnost několika dalších uzlů `alpha-test-node`. Každá cesta z uzlu `alpha-subtop-node` do uzlu `alpha-memory-node` skrze posloupnost uzlů `alpha-test-node` vyhodnocuje jednu podmínku nějakého pravidla (případně skupiny pravidel). Mějme například pravidla

[31] http://en.wikipedia.org/wiki/Directed_acyclic_graph

```
(defrule rule1
  (in :object box :location ?locaction)
  =>
  ...)
(defrule rule2
  (in :object box :location ?loc)
  =>
  ...).
```

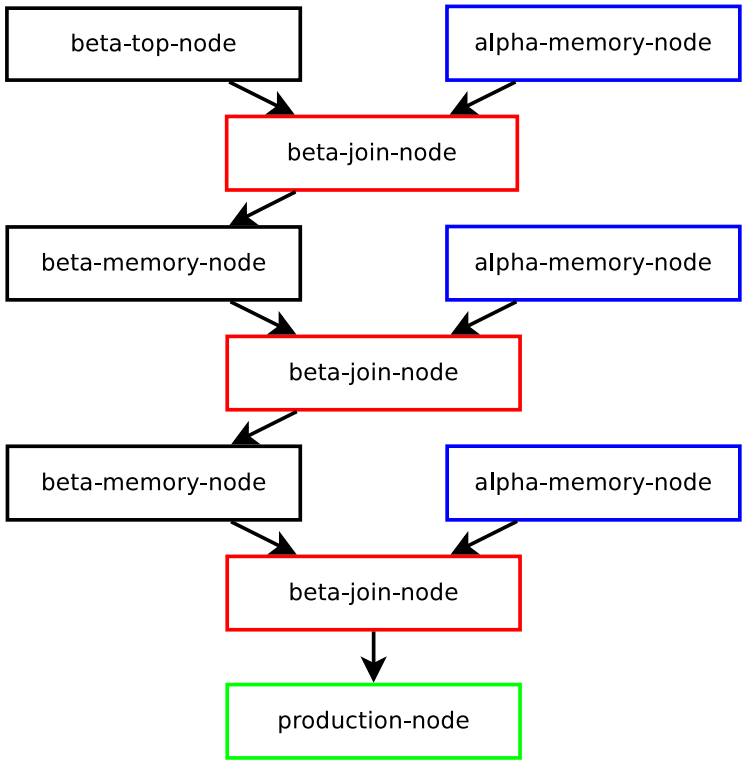
Tato pravidla mohou sdílet část alpha sítě pro vyhodnocení svých podmínek, neboť ty se liší pouze názvy proměnných. Pro vyhodnocení těchto podmínek budeme potřebovat jeden `alpha-subtop-node` pro šablonu `in`, jeden `alpha-test-node` pro testování, zda je hodnota slotu `object` rovna hodnotě `box`, a jeden `alpha-memory-node` pro uložení faktů, které tuto podmínku splňují. Pro slot `location` `alpha-test-node` nepotřebujeme, neboť alpha část sítě se hodnotami proměnných nezabývá.



Obrázek 3. Alpha část sítě RETE

Obrázek 4. zobrazuje úsek beta části sítě, spolu s uzly `alpha-memory-node`, kterými sem vstupují fakta splňující jednotlivé podmínky pravidel. Každý `beta-memory-node` udržuje množinu tokenů reprezentujících posloupnosti faktů, z nichž každý splňuje podmínku nějakého pravidla. Uzly `beta-join-node` pak

testují konzistenci vazeb proměnných mezi těmito podmínkami. `Beta-top-node` je speciální typ uzlu `beta-memory-node`, který udržuje v paměti pouze jeden prázdný token. `Production-node` je pak speciální typ uzlu `beta-memory-node`, který při aktivaci upozorní prostředí, že byly splněny všechny podmínky pravidla, případně že po odstranění faktu už nejsou všechny splněny.



Obrázek 4. Beta část sítě RETE

Mějme například pravidlo

```
(defrule rule
  (goal :object ?obj :from ?from :to ?to)
  (in :object ?obj :location ?from)
  (in :object robot :location ?to)
  =>
  ...).
```

Alpha část sítě pro toto pravidlo bude obsahovat tři uzly `alpha-memory-node`, každý pro jednu jeho podmínku (kdyby však byla v poslední podmínce místo hodnoty `robot` proměnná, vystačili bychom si se dvěma uzly `alpha-memory-node`).

Beta část sítě bude vypadat tak, jako na obrázku 4. První (shora) `beta-test-node` ve skutečnosti nemá co testovat, neboť jde o první podmínku pravidla. První `beta-memory-node` bude udržovat tokeny délky 1 reprezentující „posloupnost“ faktů, které splňují první podmínku. Druhý `beta-join-node`

bude testovat konzistenci vazeb mezi druhou a první podmínkou pravidla. Druhý `beta-memory-node` bude udržovat tokeny délky 2 reprezentující dvojice faktů, splňující první dvě podmínky pravidla. Třetí `beta-join-node` bude testovat konzistenci vazeb mezi třetí podmínkou pravidla a předchozími dvěma. `Production-node` pak bude udržovat tokeny délky 3 reprezentující trojice faktů splňující všechny podmínky pravidla.

Přidávejme nyní postupně fakty do pracovní paměti. Přidáním `(in :object box :location A)` dojde (po průchodu alpha částí sítě) k aktivaci druhého uzlu `alpha-memory-node`. Ten uloží fakt do své paměti a aktivuje „zprava“ druhý `beta-join-node`. Ten prohledá paměť uzlu `beta-memory-node` nad sebou, zda neobsahuje nějaký token s konzistentními vazbami proměnných. Ta je ale prázdná, takže zde se aktivace zastaví.

Přidáním `(goal :object box :from A :to B)` dojde k aktivaci prvního uzlu `alpha-memory-node`. Ten po uložení faktu aktivuje první `beta-join-node`. Ten nemá co testovat, takže aktivuje `beta-memory-node` pod sebou. Ten uloží jednoprvkový token s přidaným faktem a aktivuje opět druhý `beta-join-node`, tentokrát však „zleva“. `Beta-join-node` tentokrát prohledá obsah paměti uzlu `alpha-memory-node` nad sebou, zda neobsahuje fakt konzistentní s tokenem. `Beta-join-node` zde testuje, zda jsou hodnoty slotů `object` a `location` faktu v alpha paměti shodné s hodnotami slotů `object` a `from` prvního faktu tokenu v beta paměti.

Test zde skončí úspěšně, takže druhý `beta-join-node` aktivuje `beta-memory-node` pod sebou a předá mu dvouprvkový token s oběma fakty. Ten aktivuje „zleva“ třetí `beta-join-node`. Ten následně prohledá paměť uzlu `alpha-memory-node` nad sebou. Ta je ale prázdná, takže zde aktivace skončí.

Po přidání faktu `(in :object robot :location B)` dojde k aktivaci třetího uzlu `alpha-memory-node`. Ten fakt uloží a aktivuje „zprava“ třetí `beta-join-node`. Ten prohledá paměť uzlu `beta-memory-node` nad sebou, zda neobsahuje konzistentní tokeny. Ta obsahuje dvouprvkový token s prvními dvěma fakty. `Beta-join-node` tedy srovná hodnotu slotu `location` přidaného faktu s hodnotou slotu `to` prvního faktu tokenu. Protože tyto hodnoty se shodují, aktivuje `beta-join-node` `production-node` pod sebou a předá mu tříprvkový token s posloupností faktů, které splňují všechny podmínky pravidla. Ten token uloží a upozorní prostředí, že pravidlo bylo splněno posloupností faktů v tokenu.

Jak jsme viděli, `beta-join-node` může být aktivován buď zprava uzlem `alpha-memory-node` s novým faktem, který splňuje nějakou podmínku pravidla, nebo zleva uzlem `beta-memory-node` s tokenem, jehož fakty splňují předchozí podmínky. V každém případě `beta-join-node` prohledá obsah paměti druhého paměťového uzlu nad ním, aby nalezl tokeny konzistentní s novým faktem, nebo fakty konzistentní s novým tokenem.

Speciálním typem uzlu `beta-join-node` je `beta-negative-node`, který je zprava aktivován uzlem `alpha-memory-node` udržujícím fakty, které splňují ne-

govanou podmínku pravidla. Ten naopak testuje, že druhá paměť **neobsahuje** konzistentní fakty/tokeny, a v takovém případě aktivuje potomky.

Při odebrání faktu z pracovní paměti se síť chová velmi podobně. Paměťové uzly ale v tomto případě odebírají fakty/tokeny ze svých pamětí. Je-li při odebrání faktu aktivován `production-node`, upozorní prostředí, že pravidlo už není původní posloupností faktů splněno.

Odlišně se ovšem při odebrání faktu chovají uzly `beta-negative-node`. Každý `beta-negative-node` je potomkem dvou paměťových uzlů - jednoho uzlu `beta-memory-node` a jednoho uzlu `alpha-memory-node`. Pokud byl odebíraný fakt v paměti uzlu `alpha-memory-node` jediným faktem, konzistentním s nějakým tokenem v paměti uzlu `beta-memory-node`, je nyní negovaná podmínka pravidla splněna a `beta-negative-node` aktivuje `beta-memory-node` pod sebou tímto tokenem, jako by šlo o nově přidaný token.

Vzhledem k tomu, že konzistence vazeb proměnných negovaných podmínek je testována speciálním typem uzlu, je třeba, aby byly všechny proměnné negované podmínky použity v některé z předchozích pozitivních podmínek. S negovanou podmínkou není spárovaný žádný konkrétní fakt, který by se objevil v tokenu, předaném potomkům. Uzly níže v síti tedy nemohou konzistenci těchto vazeb testovat.

Kdybychom nyní přidali pravidlo, které má první dvě podmínky stejné jako předchozí pravidlo, většina beta části sítě by zůstala stejná. Druhý `beta-memory-node` by ale získal jako potomka další `beta-join-node`, který by testoval konzistenci vazeb proměnných ve třetí, odlišné podmínce vůči prvním dvěma. Pod ním by pak následovaly další uzly testující zbytek podmínek a nový `production-node`. Kdybychom přidali pravidlo, které má všechny podmínky stejné, síť by se dokonce nezměnila vůbec. Původní `production-node` by si jen zapamatoval, že jeho aktivace nyní značí splnění obou pravidel.

Každý typ uzlu sítě RETE je v programu reprezentován jedním typem objektu stejného názvu. Každý z těchto objektů implementuje metodu `activate`, jejímž zavoláním je uzel aktivován při přidání faktu do pracovní paměti a metodu `inactivate`, jejímž zavoláním je aktivován při odebrání faktu. Většina kódu implementujícího modul `rete` je tvořena právě definicemi metod `activate`. Další část kódu pak implementuje dotváření sítě při přidání pravidla, které je netriviální. Program zde prochází podmínky pravidla, přičemž zároveň postupuje v síti odshora a zkoumá, zda je třeba vytvářet nové uzly, nebo lze použít existující, případně je nově propojit.

Teorii potřebnou k implementaci algoritmu RETE jsem nastudoval z [6].

3.4.3. Kompozitní podmínky pravidel

Podporu kompozitních podmínek odvozovacích pravidel (vnořené aplikace logických funkcí v podmínkách) a všeobecné kvantifikace podmínek jsem v ExiLu neimplementoval, neboť ta by vyžadovala komplexní změny v implementaci al-

goritmu RETE, která je už tak poměrně složitá. Síť navíc při definici nových pravidel velice rychle roste a obsahuje cykly (nikoli z pohledu aktivace uzlů, ale z pohledu referencí, které na sebe uzly udržují), díky čemuž se velice těžko ladí. Alespoň tedy nastíním, z čeho bych při implementaci těchto podmínek vycházel.

[6] uvádí, jak v síti RETE implementovat tzv. konjunktivní negace, například podmínku ve tvaru

```
(and (?x on ?y)
     (?y left-of ?z)
     (not (and (?z color red)
               (?z on ?w)))).
```

Příklad pracuje s barevnými bloky, které lze stavět na sebe. Tato podmínka je splněna ve chvíli, kdy existuje blok `?x` na bloku `?y`, nalevo od bloku `?z`, který není zároveň červený a umístěn na nějakém dalším bloku. Negace a konjunkce mohou být v podmínkách libovolně vnořovány.

Pro skupinu podmínek v negaci vytvoříme podsíť uzlů, stejně jako v původním algoritmu. Místo použití jednoho `beta-negative-node` je pak pro vyhodnocení složené negace použita dvojice speciálních uzlů. První z nich se stane potomkem uzlu `beta-memory-node`, který udržuje tokeny s fakty navázanými na vzory předchozích podmínek. Druhý uzel je pak potomkem prvního uzlu z této dvojice, zároveň však i potomkem posledního uzlu v podsíti.

Druhý uzel funguje podobně, jako běžný `beta-negative-node` s tím, že ale testuje konzistenci vazeb mezi tokeny z předchozích podmínek a tokeny přicházející z podsítě. Pokud k danému tokenu shora neexistuje konzistentní token z podsítě, je konjunktivní negace na této úrovni splněna a druhý uzel aktivuje potomky.

Takto můžeme postupovat rekurzivně až na úroveň, kdy jsou negovány pouze jednotlivé podmínky. Pro ty je pak použit běžný `beta-negative-node`. Díky konjunkci a negaci, které lze libovolně vnořovat, můžeme implementovat také disjunkci (neboť $\varphi \vee \psi \Leftrightarrow \neg(\neg\varphi \wedge \neg\psi)$) a dokonce všeobecnou kvantifikaci (neboť $(\forall x)(P(x)) \Leftrightarrow (\nexists x)(\neg P(x))$).

Síť už ale nelze budovat lineárně, nýbrž je nutné stavět ji stromovitě s rekurzivním vyhodnocováním podmínek. [6] už navíc neuvádí, jak při aktivaci pravidel navazovat hodnoty proměnných. Například u disjunktivních podmínek obdrží sice `production-node` token s posloupností faktů, pokud mají ale podmínky v disjunkci stejnou strukturu, nevíme, na kterou z podmínek byl daný fakt návázán.

3.4.4. Undo/redo

Možnost vrácení provedených akcí a jejich opětovného provedení (undo/redo) je implementována na úrovni prostředí, které udržuje aktuální stav systému. Hodnoty, které stav tvoří, jsem popsal v kapitole 3.3.8. a možnosti, které undo/redo poskytuje, v kapitole 3.3.6.

Prostředí je v programu reprezentováno objektem `environment`. Stav systému je uchováván ve slotech tohoto objektu.

Prostředí dále udržuje dva zásobníky - `undo` a `redo`. Každý z těchto zásobníků ukládá seznamy tvořené popiskem akce k vrácení, uzávěrem, který akci vrátí, a uzávěrem, který ji opět provede. První uzávěr ve svém lexikálním prostředí uchovává hodnoty potřebné k vrácení akce. Zásobníky jsou manipulovány pouze několika makry a metodami k tomu určenými, zbytek kódu k zásobníkům přímo nepřistupuje.

Tělo každé akce, která mění hodnoty prostředí, je obaleno buď voláním makra `with-undo`, nebo `with-saved-slots`. Volání `with-undo` je kromě těla akce předán uzávěr, který zajistí její vrácení. Makro `with-undo` po vyhodnocení těla akce zajistí, že se na zásobník `undo` uloží předaný uzávěr spolu s uzávěrem, který akci opět provede při volání `redo`. Tento druhý uzávěr pouze vyhodnotí původní tělo akce.

Makro `with-saved-slot` zjednodušuje volání `with-undo`. Toto makro bere kromě těla akce seznam slotů prostředí, které je třeba před akcí uložit. Makro pak automaticky vytvoří uzávěr, který předá makru `with-undo`. Tento uzávěr si ve svém lexikálním prostředí pamatuje kopie původních hodnot požadovaných slotů prostředí a při vyhodnocení je nastaví zpět na tyto hodnoty.

Prostředí ke každému svému slotu definuje metodu `copy-<slot>`, např. `copy-facts`. Makro `with-saved-slots` tedy vytvoří potřebný uzávěr tak, že pomocí volání makra `let` naváže výsledky volání těchto metod pro každý požadovaný slot. V těle tohoto volání pak vytvoří anonymní (lambda) funkci, která hodnoty prostředí nastaví zpět na ty, které `let` navázal.

Funkce `undo` pak jednoduše odstraní seznam z vrcholu zásobníku `undo`, zavolá uzávěr, který vrátí poslední akci, a druhý uzávěr, který akci opět provede, uloží na vrchol zásobníku `redo`. Funkce `redo` funguje symetricky.

Všechny objekty definované modulem `core` - šablony, fakty, vzory a pravidla jsou implementovány jako neměnné (*immutable*). Každá jejich případná změna tedy znamená vytvoření nového objektu. Díky tomu není na úrovni prostředí nutné tyto objekty kopírovat. Kopírovat je třeba pouze datové struktury, které udržují kolekce těchto objektů.

Nakonec je třeba zajistit, aby každé volání `with-undo` uložilo na zásobník právě jeden uzávěr k vrácení akce. Například pokud samostatně voláme makro `assert` pro přidání nějakého faktu do pracovní paměti, chceme, aby toto volání bylo možné vrátit. Pokud ale voláme `step`, jehož výsledkem je aktivace nějakého pravidla, které ve svých důsledcích volá makro `assert`, chceme, aby se volání `step` uložilo na zásobník jako jedna akce. Volání `assert`, ke kterému v průběhu této akce dojde, už samostatně ukládat nechceme.

K tomuto účelu definuje prostředí *dynamickou proměnnou* `*undo-enabled*`. Makro `with-undo` ukládá uzávěr na zásobník jen v případě, že je hodnota této proměnná `true`. Tělo akce pak makro vyhodnocuje s hodnotou této proměnné navázanou na `false`. Tím je zajištěno, že se uzávěr na zásobník `undo` uloží vždy

jen v „nejvnějšnějším“ volání makra `with-undo`.

Díky makrům `with-undo` a `with-saved-slots` je možné velmi snadno přidat do prostředí další akce s možností jejich vrácení. Stačí jen tělo akce obalit jedním z těchto maker bez nutnosti vědět, jak je vrácení akce implementováno. Pokud potřebujeme do prostředí přidat další slot, jehož hodnotu je třeba při volání `undo` obnovovat, stačí k tomuto slotu implementovat metodu `copy`.

Nejsložitějším problémem při implementaci undo/redo bylo implementovat metodu `copy-rete`. Ta kopíruje dataflow síť RETE uloženou ve slotu `rete` prostředí. Kopírovnání sítě RETE je složité, neboť síť obsahuje cykly. Síť se sice z pohledu aktivace uzlů chová jako acyklický graf, některé uzly však uchovávájí reference na své sousedy proti směru hran tohoto grafu.

Pro účely kopírování sítě RETE jsem implementoval obecnou funkci vyššího řádu[32] pro průchod cyklickým grafem. Tato funkce využívá techniky memoizace[33]. Funkce bere kromě výchozího uzlu grafu tři funkce, které aplikuje na navštěvované uzly před memoizací a jimiž agreguje hodnoty vrácené sousedy uzlu.

Fungování funkce je pro textový popis příliš složité, považuji ji však za jednu z nejzajímavějších částí nového kódu, hlavně díky její obecnosti. Zdrojový kód funkce je v souboru `rete/graph-traversal.lisp`, kód kopírující síť rete, který funkci využívá pak v souboru `rete/rete-copy.lisp`. Celkově je kód zajišťující kopírování sítě RETE asi třikrát delší, než zbytek kódu implementující funkcionalitu undo/redo. Ten se nachází v souboru `environment/env-undo.lisp`.

3.4.5. Zpětná inference

Zpětná inference hledá odpovědi na základě zadaných cílů. Odpovědi jsou zde reprezentovány použitou substitucí proměnných v cílech a posloupností faktů a pravidel, použitých k jejich splnění. Možnosti zpětného řetězení a způsob jeho použití jsem popsal v kapitole 3.3.7.

Kód implementující zpětnou inferenci pracuje se dvěma datovými strukturami - seznamem cílů a zásobníkem pro backtracking (návrat ve výpočtu, pokud daná cesta nevede ke splnění všech cílů, nebo pro hledání alternativních odpovědí). Program nemodifikuje pracovní paměť, ani nevyhodnocuje odvozovací pravidla, pouze analyzuje volání `assert` v jejich důsledcích.

Cíle jsou v programu reprezentovány vzory, podobně jako podmínky pravidel. Inferenci si tedy můžeme představit tak, jako bychom definovali jedno pravidlo, jehož podmínky reprezentují seznam cílů a hledali všechny možné cesty výpočtu, vedoucí k jeho splnění.

Zásobník pro backtracking (dále jen zásobník) ukládá struktury, z nichž každá reprezentuje právě provedený krok výpočtu. Tato struktura je tvořena aktuálním seznamem cílů, faktem či pravidlem, použitým pro splnění aktuálního cíle, a se-

[32] http://en.wikipedia.org/wiki/Higher-order_function
[33] http://en.wikipedia.org/wiki/Memoization

znamem faktů a pravidel, která již byla pro splnění tohoto cíle použita (v případě návratu výpočtu).

Zpětná inference je řízena třemi metodami prostředí - `back-step`, `backtrack` a `back-run`. Metoda `back-step` vybere první cíl ze seznamu a hledá nejprve fakta, poté pravidla, vedoucí k jeho splnění, přičemž ignoruje ta, která už byla pro splnění cíle dříve použita.

Při hledání faktů, která splňují cíl, srovnáváme fakt se vzorem cíle podobně, jako při vyhodnocování podmínek pravidel při dopředné inferenci (viz kapitola 3.3.4.). Výsledkem tohoto srovnání je v případě shody substituce proměnných, vyskytujících se ve vzoru cíle. Je-li nalezen fakt, který aktuální cíl splňuje, program uloží nalezenou shodu na zásobník, načež cíl odstraní ze seznamu cílů a aplikuje použitou substituci proměnných na zbytek cílů. Není-li nalezen takový fakt, uvažuje program dále odvozovací pravidla.

Při hledání pravidel, která vedou ke splnění aktuálního cíle, program zohledňuje všechna volání `assert` v jejich důsledcích. Reprezentace faktů, použité ve voláních `assert`, nechá program zpracovat `parser`em a výsledné fakty pak srovnává se vzorem cíle podobně, jako fakty v pracovní paměti. Protože jde však o důsledkovou část pravidla, volání `assert` mohou obsahovat proměnné. Program tedy hledá unifikaci aktuálního cíle s tímto „proměnným faktem“.

V případě shody je výsledkem unifikace opět substituce proměnných. Program v tomto případě opět uloží shodu na zásobník, odstraní aktuální cíl ze seznamu a aplikuje použitou substituci proměnných na zbytek cílů. Nejprve ale program přidá podmínky použitého pravidla do seznamu cílů. Podmínky použitého pravidla se tedy stávají dalšími cíli, které je třeba splnit, abychom nalezli hledanou odpověď.

Hledání unifikace vzoru cíle s „proměnným faktem“ z důsledků pravidla je podobné jako v jazyce Prolog. Je však jednodušší o to, že symbolická reprezentace faktů a vzorů je lineární (seznamy nejsou vnořené) a není třeba rozlišovat mezi relačními a funkčními symboly. V případě strukturovaných faktů a vzorů srovnáváme hodnoty jednotlivých slotů, jako by šlo o atomy seznamu, jen zde nezáleží na pořadí, nýbrž srovnáváme odpovídající sloty faktu a vzoru.

Pokud metoda `back-step` nenalezne fakt ani pravidlo vedoucí ke splnění aktuálního cíle, volá metodu `backtrack`. Ta odstraňuje postupně struktury z vrcholu zásobníku, obnoví cíle do stavu při uložení struktury a hledá shody s dosud nepoužitými fakty a pravidly. Pokud takovou shodu nalezne, ubírá se výpočet dále touto cestou. Je-li celý zásobník vyčerpán před nalezením shody, nahlásí metoda neúspěch.

Metoda `back-run` volá opakovaně metodu `back-step`, dokud nejsou splněny všechny cíle, nebo není nahlášen neúspěch. V případě, že už jsou všechny cíle splněny, volá `back-step` metodu `backtrack` pro nalezení alternativních odpovědí. To může vést k nalezení nové shody, nebo vyčerpání zásobníku, což znamená, že už žádné další odpovědi neexistují.

Dospěje-li volání `back-run` ke splnění všech cílů, vytiskne se seznam faktů

a pravidel, použitých ke splnění jednotlivých cílů. Dále se vytiskne substituce proměnných, která byla pro splnění cílů použita. Tato substituce je tvořena složením všech substitucí, použitých v jednotlivých krocích výpočtu. Ve výsledné substituci jsou ponechány pouze proměnné, které uživatel použil v definicích původních cílů. Průběžné proměnné, které se sem dostaly při aplikaci odvozovacích pravidel, nejsou pro uživatele zajímavé.

Implementace zpětné inference nevyužívá sítě RETE, neboť ta vyhodnocuje splnění podmínek odvozovacích pravidel fakty v pracovní paměti. Zpětná inference ale pracuje opačně - analyzuje důsledky pravidel a pracovní paměť nemodifikuje. Síť RETE tedy nelze k implementaci využít.

Implementace je inspirována prohledáváním SLD-stromů používaným v jazyce Prolog[34]. Z toho také vyplývají její omezení. Implementace pracuje pouze s „pozitivní znalostí“ - neumožňuje zadání negativních cílů, ani negované podmínky pravidel. Implementace také uvažuje pouze přidání nových faktů v důsledcích pravidel, nikoli jejich odebrání nebo změnu. Díky tomu, že se důsledky pravidel nevyhodnocují, nedochází navíc k vedlejším efektům jako při dopředné inferenci.

Uvažovat negativní cíle, negované podmínky pravidel a odebírání faktů v důsledcích pravidla se na první pohled nezdá implementačně náročné. Nabízí se několik možností, kterak vyhodnocovat odebrání faktu v důsledcích pravidla, každá z nich však vede k určitému typu problémů.

První možností je zohledňovat volání `retract` v důsledcích pravidla jen jako indikaci, že toto pravidlo vede ke splnění negativního cíle, jinak je ignorovat. Zde ovšem hrozí problém, že důsledky některého z pravidel zneplatní podmínky dalšího. Uvažujme následující zadání:

```
(defgoal (at home))

(deffacts initial
  (out of city)
  (have money))

(defrule take-bus-to-city
  (have money)
  (out of city)
  =>
  (assert (at city))
  (retract (have money)))

(defrule take-taxi-home
  (have money)
  (at city)
  =>
```

[34] http://en.wikipedia.org/wiki/SLD_resolution#SLD_resolution_strategies

```
  (assert (at home))
  (retract (have money)).
```

Dopředná inference v tomto případě korektně selže v odvození faktu `(at home)`, neboť aplikace pravidla `take-bus-to-city` odstraní fakt `(have money)`, tudíž druhé pravidlo už nemůže být splněno. Zpětná inference by však postupovala následovně:

```
(goals)      ;=> ((at home))
(back-step)  ; use rule take-taxi-home
(goals)      ;=> ((have money) (at city))
(back-step)  ; use fact (have money)
(goals)      ;=> ((at city))
(back-step)  ; use rule take-bus-to-city
(goals)      ;=> ((have money) (out of city))
(back-step)  ; use fact (have money)
(goals)      ;=> ((out of city))
(back-step)  ; use fact (out of city)
(goals)      ;=> () => solution found
```

Nalezeným řešením je tedy postupná aplikace pravidel `take-bus-to-city` a `take-taxi-home` (zpětná inference je aplikuje v opačném pořadí), přestože toto řešení je nesprávné.

Další možností, která se nabízí, je pamatovat si všechny cíle, které se někdy vyskytovaly v množině cílů, a brát volání `(retract (have money))` v důsledcích pravidla jako indikaci, že pravidlo nelze použít, pokud je cíl `(have money)` v této množině. To sice řeší předchozí problém, ale vede k dalšímu. Představme si, že bychom do znalostní báze přidali pravidlo, které umožňuje použití bankomatu:

```
(defrule use-atm
  (have card)
  (have money on account)
  =>
  (assert (have money))
  (retract (have money on account))).
```

Dopředná inference tentokrát nalezne korektní řešení postupnou aplikací pravidel `take-bus-to-city`, `use-atm`, `take-taxi-home`. Zpětná inference však selže, neboť pravidlo `take-bus-to-city` není díky tomuto způsobu vyhodnocování `retract` nikdy aktivováno.

Z neúspěchu předchozích variant je evidentní, že volání `retract` musí nějakým způsobem manipulovat množinu faktů. V jakou chvíli by k tomu ale mělo docházet? Pokud bychom fakt `(have money)` odstranili hned při aplikaci pravidla `take-taxi-home` (které zpětná inference uvažuje jako první), zneplatníme

tím jeho vlastní podmínky. Po splnění všech jeho podmínek je už ale na odstraňování faktu pozdě, neboť to už bude aplikováno i pravidlo `take-bus-to-city`, k čemuž by vůbec dojít nemělo.

K podobným problémům bude docházet při zneplatnění negativního cíle (který se do množiny cílů může dostat také jako negovaná podmínka aplikovaného pravidla) voláním `assert`. Volání `modify` dokonce tato dvě spojuje. I kdybychom navíc našli vhodný způsob, kterak množinu faktů zpětně manipulovat, možnost backtrackingu by vyžadovala ukládat průběžné množiny faktů na zásobník, který by ve větších systémech rostl neúměrně rychle. Implementace plnohodnotného zpětného řetězení tedy není tak jednoduchá, jak se na první pohled jeví, a domnívám se, že spolu s ostatními body zadání je nad rámec práce.

3.4.6. Kompatibilita se systémem CLIPS

Syntaktickou kompatibilitu se systémem CLIPS zajišťuje nově přidaný modul `parser`. Ten převádí externí reprezentace objektů - šablon, faktů, vzorů a pravidel - na interní objekty (ve smyslu instance třídy) definované v modulu `core`. K tomuto účelu definuje `parser` metodu `parse-<object>` pro každý typ `core` objektu. Externími reprezentacemi jsou (případně vnořené) seznamy symbolů, které předáváme funkcím a makrům modulu `front-end`, jež jsem popsal v uživatelské příručce.

Metody `parseru` jednoduše analyzují tyto seznamy, rozhodují, zda jde o základní, nebo CLIPSovou syntax, a podle toho je převádí na jednotnou reprezentaci, kterou pak předávají konstruktorům `core` objektů. Tyto vnitřní objekty jsou pak metodami `parseru` vraceny. Metody jsou volány modulem `front-end`, který pak získané `core` objekty předává prostředí.

3.4.7. Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je implementováno použitím knihovny CAPI[35]. Program definuje pět rozhraní (oken) voláním makra `define-interface` knihovny. Každé rozhraní je pak reprezentováno objektem `interface`.

Každý z objektů rozhraní udržuje referenci na prostředí, s nímž je rozhraní svázáno, a implementuje callbacky pro každé ze zobrazených tlačítek. Tyto callbacky pak volají příslušné funkce prostředí. Objekty rozhraní, která zobrazují seznamy hodnot prostředí, navíc implementují metodu `update-lists`, která slouží k překreslení seznamů, a dotazuje se prostředí na jejich obsah.

Každé prostředí také udržuje referenci na rozhraní, které je s ním svázáno, a upozorňuje jej na změny ve svém stavu voláním jeho metody `update-lists`.

Kód definující uživatelské rozhraní a volání `update-lists` z prostředí se vyhodnocuje jen v případě, že je kód načten v prostředí LispWorks. To zajišťuje použití direktivy `#+lispworks` před voláním příslušných funkcí.

[35] http://www.lispworks.com/products/capi.html

3.5. Referenční příručka

V uživatelské příručce jsem uvedl všechny možnosti, které knihovna ExiL poskytuje. Zde tedy uvedu pouze seznam signatur funkci a maker s odkazy na opovídající sekce příručky, pro snadnou referenci.

Definice šablon - 3.3.2., 3.3.11.

```
(defmacro deftemplate (name &body slots))
(defun deftemplatef (name slots))
(defmacro undeftemplate (name))
(defun undeftemplatef (name))
(defun templates ())
(defmacro find-template (name))
(defun find-templatef (name))
```

Definice skupin faktů - 3.3.2., 3.3.11.

```
(defmacro deffacts (name &body fact-specs))
(defun deffactsf (name fact-specs))
(defmacro undeffacts (name))
(defun undeffactsf (name))
(defun fact-groups ())
(defmacro find-fact-group (name))
(defun find-fact-groupf (name))
```

Definice pravidel - 3.3.2., 3.3.11.

```
(defmacro defrule (name &body body))
(defun defrulef (name body))
(defmacro undefrule (name))
(defun undefrulef (name))
(defun rules ())
(defmacro find-rule (name))
(defun find-rulef (name))
(defun agenda ())
```

Modifikace pracovní paměti - 3.3.3.

```
(defun facts (&optional start-index end-index at-most))
(defmacro assert (&rest fact-specs))
(defun assertf (&rest fact-specs))
(defmacro retract (&rest fact-specs))
(defun retractf (&rest fact-specs))
(defun retract-all ())
(defmacro modify (fact-spec &rest mod-list))
(defun modifyf (fact-spec mod-list))
```

Sledování průběhu inference - 3.3.5.

```
(defmacro watch (watcher))
(defun watchf (watcher))
(defmacro unwatch (watcher))
(defun unwatchf (watcher))
(defmacro watchedp (watcher))
(defun watchedpf (watcher))
```

Definice cílů - 3.3.7.

```
(defmacro defgoal (goal-spec))
(defun defgoalf (goal-spec))
(defmacro undefgoal (goal-spec))
(defun undefgoalf (goal-spec))
(defun goals ())
```

Spuštění inference - 3.3.4., 3.3.7.

```
(defun step ())
(defun halt ())
(defun run ())
(defun back-step ())
(defun back-run ())
```

Reset prostředí - 3.3.8.

```
(defun clear ())
(defun reset ())
(defun complete-reset ())
```

Undo/redo - 3.3.6.

```
(defun undo ())
(defun redo ())
(defun undo-stack ())
(defun redo-stack ())
```

Práce s více prostředími - 3.3.9.

```
(defmacro defenv (name &key redefine))
(defun defenvf (name &key redefine))
(defmacro undefenv (name))
(defun undefenvf (name))
(defmacro setenv (name))
(defun setenvf (name))
(defun environments ())
(defun current-environment ())
```

Inferenční strategie - 3.3.4.

```
(defmacro setstrategy (name))
(defun setstrategyf (name))
(defun current-strategy ())
(defun strategies ())
```

Grafické uživatelské rozhraní - 3.3.12.

```
(defun show-gui (&optional environment))
```

4. Závěr

Ke knihovně ExiL jsem implementoval možnost vrácení a opětovného provedení všech akcí s vedlejším efektem, včetně inferenčních kroků. Tato funkcionalita je kompletní, implementace je však neefektivní. V každém kroku se například kopíruje celá síť RETE. Ukládat pouze inverzní volání však není možné, neboť to vede ke změně pořadí ukládaných hodnot stavu systému, tudíž ke zvolení jiné výpočetní větve po navrácení.

Systém podporuje základní syntaxi systému CLIPS. Některé konstrukty však nejsou podporovány, protože buď v Common Lispu nemají smysl, nebo by vyžadovaly rozsáhlé úpravy vnitřní implementace systému.

Zpětné řetězení je implementováno v omezené podobě, založené na prohledávání SLD-stromů v jazyce Prolog. Stejně jako Prolog nepodporuje zpětné řetězení práci s negací a odstraňování faktů z pracovní paměti systému. Podpora této funkcionality není problematická implementačně, nýbrž principiálně. Problémy, ke kterým dochází, jsem popsal v kapitole 3.4.5.

Grafické uživatelské rozhraní je funkční, umožňuje však pouze prohlížení stavu systému a odebírání jednotlivých definic, nikoli jejich přidávání. To by znamenalo přidat textová definiční pole do jednotlivých rozhraní. K tomu ale nevidím důvod, neboť definice lze zadávat pomocí REPLu (Listeneru) interpreteru Common Lispu, přičemž se grafické rozhraní automaticky obnoví. Rozhraní je svázáno s prostředím systému, lze tedy používat více prostředí zároveň a ke každému zobrazit grafické rozhraní.

Podpora kompozitních podmínek není implementována vůbec. Podpora by vyžadovala rozšíření implementace algoritmu RETE, která je už tak složitá. Síť RETE navíc velmi rychle roste a obsahuje cykly, v důsledku čehož se velmi těžko ladí. V kapitole 3.4.3. ale popisuji, jak by tuto funkcionalitu bylo možné implementovat.

Kromě podpory kompozitních podmínek a některých složitějších konstruktů systému CLIPS, jako například multisloty, by bylo zajímavé implementovat podporu použití objektů CLOSu jako faktů expertního systému. Knihovna by také mohla podporovat například uživatelsky definované testy v podmínkách pravidel, jako porovnávání číselných hodnot, nebo rozhodování na základě návratových hodnot volání lispových funkcí. Implementace této funkcionality by však byla velmi složitá.

Zajímavá by byla také podpora nejistoty u faktů a odvozovacích pravidel systému, jakou poskytuje například systém MYCIN[36]. Tu je už teď možné nasimulovat připojením hodnot k faktům a jejich manipulací použitím funkčních alternativ maker modifikujících pracovní paměť v důsledcích pravidel. Kromě rovnosti však není možné tyto hodnoty testovat v podmínkách pravidel, navíc je tento nepřímý způsob nepohodlný.

[36] http://en.wikipedia.org/wiki/Mycin

Funkčnost systému jsem testoval na mnoha malých příkladech, které jsem buď vytvořil od základu, nebo převzal z [7] a ručním voláním funkcí a maker systému. Vytvoření rozsáhlejších příkladů nebo jejich přepsání z programů vytvořených pro systém CLIPS by bylo časově náročné. Kód programu je navíc opatřen velkou sadou integračních a unit testů.

Seznam použité literatury

[1] Kaláb, J.: *Implementace expertního systému s dopředným řetězením*. Bakalářská práce, Universita Palackého v Olomouci, 2010.

[2] Jackson, P.: *Introduction to Expert Systems*. Addison Wesley, 1998, ISBN 0-201-87686-8.

[3] Problem domain — Wikipedia, The Free Encyklopedia. 2013. http://en.wikipedia.org/wiki/Problem_domain

[4] STRIPS — Wikipedia, The Free Encyklopedia. 2012. http://en.wikipedia.org/wiki/STRIPS

[5] CLIPS: *Rereference Manual: Volume I, Basic Programming Guide*. 2007. http://clipsrules.sourceforge.net/documentation/v630/bpg.pdf

[6] Doorenbos, R. B.: *Production Matching for Large Learning Systems*. Dizertační práce, Carnegie Mellon University, 1995. http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf

[7] Norvig, P.: *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, 1991, ISBN 1-55860-191-0.

A. Obsah přiloženého CD

`doc/`

Adresář obsahuje text diplomové práce ve formátu PDF spolu se zdrojovými kódy a dodatečnými soubory pro jeho sestavení typografickým systémem LaTeX.

`src/`

Adresář obsahuje zdrojový kód implementace knihovny ExiL. Ve podadresáři `examples` je několik dalších příkladů použití knihovny.