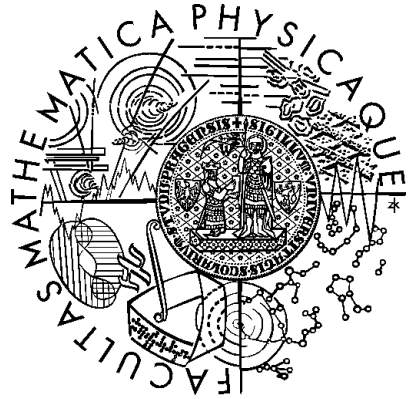
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Michal Drbohlav

GUI pro dokazovací systémy

Katedra algebry

Vedoucí bakalářské práce: **RNDr. David Stanovský Ph.D.**
Studijní program: **Informatika, obecná informatika**

2007

Na tomto místě bych rád poděkoval RNDr. Davidu Stanovskému, Ph.D. za zajímavé a podnětné téma bakalářské práce a stejně tak za její vedení. Nemohu ani opomenout tvůrce systémů automatického dokazování, bez kterých by tato práce nemohla vzniknout. Zvláště bych chtěl poděkovat Thomasi Hillenbrandovi, jednomu z tvůrců Waldmeisteru, za ochotnou pomoc při zprovozňování Cygwin verze.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 21. května 2007 Michal Drbohlav

Obsah

Název práce: GUI pro dokazovací systémy
Autor: Michal Drbohlav
Katedra: Katedra algebry
Vedoucí bakalářské práce: RNDr. David Stanovský, Ph.D.
e-mail vedoucího: David.Stanovsky@mff.cuni.cz

Abstrakt Práce popisuje program *ATP GUI*, tedy grafické prostředí pro práci se systémy automatického dokazování matematických vět. Program je primárně určen pro operační systém GNU/Linux, avšak podporován je i systém Windows XP.
K vytvoření ATP GUI byl použit programovací jazyk C s využitím knihoven Gtk+2 a libXML2. Práce dále obsahuje základní popis cílů dokazovacích systémů, přehled a krátké představení podporovaných systémů, Prover9, Otter, Mace4, Waldmeister, Paradox, E a instrukce k integrování dalších dokazovacích programů.
Součástí práce je i CD příloha obsahující samotný program včetně všech rozšíření pro operační systémy GNU/Linux a Windows, spolu s knihovnou TPTP verze 3.2.0 a elektronickou podobou této práce.

Klíčová slova: automatické dokazování, GUI, C

Title: GUI for Automatic Theorem Provers
Author: Michal Drbohlav
Department: Department of Algebra
Supervisor: RNDr. David Stanovský, Ph.D.
Supervisor's e-mail address: David.Stanovsky@mff.cuni.cz

Abstract This work describes *ATP GUI*, graphical user interface for automatic theorem provers of mathematical theorems. The programme was primarily developed for GNU/Linux operating system but can be also run under the Windows XP environment.
To create ATP GUI, C programming language with Gtk+ and libXML2 libraries were used. The work also contains basic description of automatic theorem provers' functions and goals, a brief introduction to the supported ATP systems, Prover9, Otter, Mace4, Waldmeister, Paradox, E, and instructions to be followed in order to create a new ATP GUI plug-in for other ATP systems.
This work also contains a CD attachment featuring the programme itself including all plug-ins along with the TPTP library version 3.2.0 and the electronical version of this document.

Keywords: automatic theorem proving, GUI, C

Kapitola 1

Úvod

Na úvod práce se budu věnovat základnímu teoretickému pozadí systémů automatického dokazování a popisu některých principů, které za nimi stojí.

1.1 Matematická logika

1.1.1 Základní definice

Většina systémů automatického dokazování pracuje s tzv. logikou prvního řádu. Jazyk logiky prvního řádu obsahuje:

- Neomezeně mnoho proměnných.
- Predikátové symboly (např.: $\geq, =$...). Každému predikátovému symbolu je přidělena arita (např.: *unární*, *binární*, ...). Většina systémů automatického dokazování pracuje s rovností, a tedy definuje speciální predikát rovnosti - vzniká tak predikátová logika prvního řádu s rovností.
- Funkční symboly s danou aritou.
- Logické spojky - $\rightarrow$ *implikace*, $\leftrightarrow$ *ekvivalence*, & *konjunkce*, | *disjunkce* a ! *negace*. Negace se též značí $\neg$ nebo $\sim$, ale zde zachovávám značení z ATP GUI.
- Všeobecný kvantifikátor $\forall$ - v ATP GUI značený `V`.
- Existenční kvantifikátor $\exists$ - v ATP GUI značený `E`.
- Závorky (,), {, }, [,]

Tvrzení jsou formálně reprezentována jako *formule*. Ta vznikne pomocí konečného počtu použití následujících pravidel

- Proměnná je term.
- Pokud f je n-ární funkční symbol a $t_1, ..., t_n$ jsou termy, potom $f(t_1, ..., t_n)$ je term.
- Pokud $A_1, A_2, ..., A_n$ jsou termy a p je n-ární predikátový symbol, potom $p(A_1, ..., A_n)$ je formule. Zde platí malá výjimka pro $=$ (rovnost) a $\neq$ (nerovnost), které se obvykle (nejen z historických důvodů) značí infixově, avšak i přesto spadají pod toto pravidlo. Totéž může platit i u jiných predikátů ($<$ se typicky používá infixově) nebo i funkčních symbolů ($+$, ...). Formule v tomto tvaru se nazývá *atomická*.
- Pokud F je formule, potom (F), $[F]$ i $\{F\}$ jsou formule.
- Pokud F je formule, potom $!F$ je formule.
- Pokud F a G jsou formule a X je logická spojka kromě ! (ve smyslu předchozí definice), potom $F\ X\ G$ je formule.
- Pokud F je formule, Q je jeden z kvantifikátorů a v je proměnná označující individuum (tedy například prvek grupy - obecně prvek domény, na které formule žijí, ale ne například zobrazení), potom $(Qv)F$ je formule.

O formuli řekneme, že je *otevřená*, jestliže žádná z jejích proměnných není kvantifikována, naopak, formule je *uzavřená*, jestliže jsou všechny proměnné kvantifikovány.

Celý formální systém potom tvoří jazyk, vybrané formule, označené jako *axiomy* - ty reprezentují tvrzení, o kterých se předpokládá, že jsou pravdivá. Poslední součástí formálního systému jsou tzv. odvozovací pravidla, která z axiomů (a případně jiných, již odvozených, formulí) odvodí další, o kterých se také soudí, že jsou pravdivé. Nejběžnějším příkladem odvozovacího pravidla je *Modus ponens*:

$$\frac{A \to B \quad A}{B}$$

(toto je ustálené značení odvozovavích (resolučních) pravidel, nad čarou se vyskytují předpoklady, pod ní formule, kterou lze odvodit)

Důkazem rozumíme takovou konečnou posloupnost formulí $\{A_k\}_{k=1}^n$, kde A_n je dokazovaná formule a každá formule A_k je bud' axiom, nebo vznikne odvozením pomocí nějakého pravidla z předchozích formulí. O formuli řekneme, že je dokazatelná (v daném formálním systému), pokud existuje nějaký důkaz.

Realizací jazyka rozumíme relační strukturu:

- Neprázdná množina individuí M, též nazývaná doména.
- Pro každý n-ární funkční symbol f zobrazení $f : M^n \to M$
- Pro každý n-ární predikátový symbol kromě symbolu pro rovnost relaci $p \subseteq M^n$.

Modelem pak rozumíme takovou realizaci jazyka, která splňuje všechny axiomy dané teorie. Každá bezesporná teorie má nějaký model. Podle Gödelovy věty jsou v dané teorii T dokazatelné právě ty formule, které jsou splněné v každém modelu T. Zkoumat všechny modely teorie sice není možné (často už proto, že jich není konečný počet), ale i tak se tohoto dá dobře využít při hledání protipříkladů, stačí totiž najít jediný model, v němž dokazovaná formule není splněná.

Zvláštní postavení mají mezi teoriemi tzv. *úplné teorie*. Úplná je taková teorie T, která je bezesporná a pro každou formuli A v jazyku T platí, že bud' A, nebo $!A$ je dokazatelná. Takové teorie avšak nejsou příliš časté - například predikátová logika s rovností takové vlastnosti nemá (například formule $x = 3$ i $x \neq 3$ jsou nedokazatelné).

1.1.2 Skolemizace

Většina systémů automatického dokazování neoperuje s celým jazykem 1. řádu; vyžaduje axiomy a dokazované formule v určitém tvaru. Jen velice málo systémů je schopno operovat s explicitním kvantifikováním proměnných. Systém pracuje s formulemi, které jsou univerzálně kvantifikovány přes všechny proměnné.

Některé systémy navíc vyžadují, aby formule byly v tzv. *Konjunktivně normálním tvaru* (CNF):

- Formule je v *konjunktivním normálním* tvaru, jestliže je konjunkcí *klausulí*.
- *Klausule* je disjunkce *literálů*.
- *Literál* je A nebo $!A$, kde A je atomická formule.

Tyto požadavky ovšem nijak nezmenšují možnosti specifikace problémů, jelikož každou logickou formuli lze převést na tzv. *Skolemův normální tvar* a jeho jádro lze převést do CNF.

Formule je v prenexním tvaru, jestliže má tvar

$$(Qx)(Qy)...B,$$

kde Q je kvantifikátor a B, někdy označované jako jádro formule, žádné kvantifikátory neobsahuje. Každá formule lze pomocí jednoduchých pravidel převést do prenexního tvaru. O formuli řekneme, že je ve *Skolemově normálním tvaru*, jestliže navíc všechny kvantifikátory jsou všeobecné.

K převedení do tohoto tvaru slouží postup, který se nazývá *skolemizace*. Ke každé formuli ve tvaru

$$(\forall x_1)(\forall x_2)...(\forall x_n)(\exists y)...A(x_1, x_2, ..., x_n, y, ...)$$

existuje funkce f taková, že formule

$$(\forall x_1)(\forall x_2)...(\forall x_n)...A(x_1, x_2, ..., x_n, f(x_1, x_2, ..., x_n), ...)$$

je splněna právě tehdy, když je splněna původní formule. Z čistě formálního hlediska se nejedná o tytéž formule, ale z hlediska splnitelnosti jsou ekvivalentní. Funkční symbol f se nazývá *Skolemova funkce*. Takto lze indukčně získat formule ve Skolemově normálním tvaru. Nevýhodou tohoto je však fakt, že tento postup nekonstruuje vytvořená zobrazení, pouze konstatuje jejich existenci. Jinými slovy, do jazyka se zavádí funkční symboly, o kterých nikdo neví, jak se chovají. Nicméně toto již poskytuje dobrý základ, ke tvrzení samotné *Skolemově věty*, která tvrdí, že ke každé teorii 1. řádu existuje ekvivalentní otevřená teorie, tedy taková, jejíž axiomy jsou otevřené formule. V této situaci vlastně stačí věta o uzávěru[14].

Takto je vidět, že výše uvedené požadavky některých systémů nijak nezmenšují prostor řešitelných problémů. Ve skutečnosti i systémy jako Prover9[1], které kvantifikaci podporují, převedou samy problém do otevřené teorie. algoritmicky se nejedná o obtížnou záležitost, nicméně pro ATP GUI jsem se rozhodl ji neimplementovat, právě z výše uvedeného důvodu nekonstruktivnosti skolemizace.

1.2 Automatické dokazování

Na této teorii jsou potom postaveny systémy automatického dokazování. Budu se zabývat jen plně automatickými systémy, existují i interaktivní, které při běhu

vyžadují od uživatele nápovědy. Ty pracují většinou v logice druhého řádu, která je bližší tradičnímu způsobu matematické argumentace. Plně automatické systémy operují s logikou prvního řádu, někdy i ochuzenou o kvantifikátory. U některých oborů (teorie grup, množin...) toto omezení nepůsobí problémy, ale obory jako analýza, teorie čísel nebo geometrie jsou nevhodné. Veškerá matematická teorie jde sice specifikovat v axiomech prvního řádu přes teorii množin, avšak výpočetně je to nezvladatelné. I přesto se v manuálu systému Otter[2] najde zmínka, že pomocí tohoto systému se podařilo dokázat některé zajímavé věty z geometrie a teorie čísel. Omezení na logiku prvního řádu na druhou stranu přináší některé výhody. Patrně nejzřejmější je jednoduchost jejího jazyka.

Složitost automatického dokazování v závislosti na použitém logickém základu sahá od triviálních problémů (například pokud je množina axiomů sporná, tak lze dokázat úplně cokoliv) až po algoritmicky neřešitelné problémy. Ani omezení na první řád však nezaručuje úspěch; V důsledku Druhé Gödelovy věty je jakákoliv bezesporná teorie obsahující elementární aritmetiku nerozhodnutelná. To vylučuje možnost existence nějakého univerzálního algoritmu dokazujícího tvrzení v logice 1. řádu. Aritmetika navíc není v logice 1. řádu konečně axiomatizovatelná, axiom indukce

$$A_x[0] \rightarrow ((\forall x)\,(A \rightarrow A_x[S(x)]) \rightarrow (\forall x)A)\,,$$

kde A je formule z jazyka aritmetiky a $S(x)$ značí následníka prvku x, nelze specifikovat v logice 1. řádu. I přes teoretická omezení však systémy automatického dokazování dokáží řešit i poměrně složité problémy.

Systémy se tedy ocitají v poměrně nezáviděníhodné pozici, kdy se od nich očekává, že budou řešit algoritmicky neřešitelný problém. Nenajdeme tedy systém, který by se snažil být příliš obecný (taková snaha by byla odměněna zaslouženým neúspěchem, právě kvůli platnosti Druhé Gödelovy věty o nerozhodnutelnosti), naopak se setkáváme s masivní specializací (např. Waldmeister operuje jen s jednotkovými klauzulemi, kde navíc každá klauzule je rovnost) a systém bez ohledu na zadanou teorii dokazuje jedním způsobem. Oblíbenými způsoby jsou hlavně důkaz sporem, hledání protipříkladu nebo Knuth-Bendixův algoritmus.

1.3 Knuth-Bendixův algoritmus

Knuth-Bendixův algoritmus[15] vytvoří přepisovací systém z množiny rovností. Jedná se o základní postup při řešení problému hledání důkazu, všechny systémy, vyjma těch, které hledají modely teorií, jsou na něm založeny (Waldmeister[4]) nebo alespoň využívají některé jeho prvky (např. Prover9[1] nebo E[6] využívají tzv. *superposition calculus*), třebaže se nejedná o jeho původní podobu z roku 1970. Původní verze může skončit úspěchem, neúspěchem, nebo nemusí skončit vůbec. V roce 1989

byl publikován algoritmus Unfailing Knuth-Bendix Completion[16], který vylučuje druhou možnost, a právě na této verzi je založen systém Waldmeister[4].

Algoritmus má následující předpoklady:

- Necht' T je teorie sestávající se z rovností mezi termy.
- Necht' $<$ je relace dobrého uspořádání[1] na termech.
- Dále necht' platí $x_i < y_i \Rightarrow t(x_1, ..., x_n) < t(y_1, ..., y_n)$.

Relační struktura na struktuře termů je definovaná pomocí rovností, tedy je to ekvivalence, a proto lze množinu termů faktorizovat. Dva termy jsou si rovny, pokud patří do stejné třídy ekvivalence. Dokazované tvrzení je rovnost dvou termů a pro oba termy chceme najít nejmenší prvek jeho třídy ekvivalence. Pokud pro oba termy z rovnosti nalezneme toho samého reprezentanta třídy, dokázali jsme, že termy jsou si v teorii T rovny.

K hledání nejmenšího reprezentanta třídy použijeme přepisovací systém vytvořený z axiomů teorie T a daný term budeme přepisovat tak dlouho, dokud půjde pokračovat dál. Výsledkem přepisování je právě reprezentant, kterého hledáme.

Pro vytvoření přepisovacího systému je nejprve třeba takzvaně zorientovat rovnosti, neboli vytvoříme z rovností taková pravidla, že větší term se přepíše na menší. Triviální rovnosti sice orientovat nelze, ale bez újmy je můžeme ignorovat. Dále hledáme tzv. *kritické páry*. Pokud lze daný term přepsat pomocí dvou různých pravidel, potom je nutné zajistit, aby oba výsledky (nazvěme je v_1 a v_2) šly dále přepsat do společného tvaru. V případě, že to není zajištěno, potom právě označujeme výše uvedenou dvojici pravidel jako kritický pár. K nápravě tohoto kritického páru stačí přidat nové pravidlo $\max v_1, v_2 \rightarrow \min v_1, v_2$ (pokud $v_1 \neq v_2$). Takto přidáváme nová pravidla, dokud existují nějaké nevyřešené kritické páry. Je vidět, že takto vzniklý přepisovací systém nemusí být konečný, a tedy algoritmus nemusí skončit.

1.4 Praktické výsledky automatického dokazování

Automatickým dokazováním by se samozřejmě nikdo nezabýval, pokud by neneslo žádné výsledky. Jako první oblast se nabízí dokazování otevřených matematických problémů. Asi nejznámějším výsledkem z této oblasti bylo vyřešení *Robbinsova problému*, tedy ekvivalence mezi *Robbinsovými* a *Boolovskými* algebrami. Tento problém byl otevřený od roku 1933 do poloviny 90. let 20. století, kdy byl automaticky vyřešen systémem EQP. Dalším příkladem mohou být některá tvrzení v teorii

[1] Tedy je lineání a každá podmnožina množiny termů má nejmenší prvek

kvazigrup. Dále se setkáme s nalezením odlišné axiomatizace nějaké struktury, často se jedná o jediný axiom. Taková reprezentace je sice pro člověka nepřehledná, ale počítačům se s ní naopak může pracovat lépe, většinou to je ale spíš jen zajímavost.

Mimo to je automatické dokazování též základem testování korektnosti hardwaru i sotfwaru. Například po zkušenostech se slavným FDIV bugem u prvních Pentií je pomocí dokazovacích systémů ověřována funkčnost FPU[10]. Na univerzitě Koblenz-Landau v Německu například vyvíjí systém automatického průvodce, který turistům umí přes Bluetooth posílat na mobilní telefon informace o různých památkách a jiných pamětihodnostech. Turista si nainstaluje program, který tyto informace přijímá a pomocí systému automatického dokazování se na základě turistou definovaného profilu rozhoduje, jestli informace je relevantní nebo ne a bud' ji turistovi zobrazí, nebo ji zahodí.

Kapitola 2

Systémy automatického dokazování

V této kapitole se budu věnovat jednotlivým systémům podporovaným ATP GUI a některým dalším, na které jsem narazil během zíkávání informací pro tuto práci, a které se mi zdály z nějakého důvodu zajímavé.

Na CD, které je přílohou této práce, se nachází balíčky pro integraci 6 systémů do ATP GUI. Všechny systémy kromě Waldmeisteru jsou distribuovány pod GNU GPL[9] verze 2, proto bylo možné je zahrnout do obsahu CD. Systém Waldmeister se ovšem vzhledem k oficiální povaze práce na CD vyskytovat nemůže. Pro jeho získání je nutno navštívit stránky projektu *http://www.waldmeister.org/*.

2.1 Prover9 a Otter

Prover9[1] je systém automatického dokazování prvního řádu s rovností. Systém používá mnoho pravidel, největší význam z nich má paramodulace. Stejně jako jeho přímý předchůdce Otter[2] (právě Otter jako první použil paramodulaci) dokazuje tvrzení zásadně sporem. Tedy jako většina kalkulů prvního řádu se snaží ukázat nesplnitelnost množiny klausulí, v tomto případě složenou z axiomů a negovaného dokazovaného tvrzení. Pro každou takovou nesplnitelnou množinu lze ukázat její nesplnitelnost (právě díky výše zmíněné paramodulaci). To ovšem nijak nezaručuje, že ten postup algoritmus najde. V praxi je program navíc omezen množstvím prostředků a rozumnou dobou výpočtu.

Prover9 jako jeden z mála systémů podporuje celý jazyk logiky prvního řádu (včetně kvantifikátorů), implementuje automaticky skolemizaci, převod do CNF,

obsahuje předdefinované predikáty, operátory, umožňuje forward- a back- demodulation, forward- a back- supsumption, vážení formulí. Implementuje některá pokročilejší inferenční pravidla, například

- binární resoluce

$$\frac{\neg P(X,Y) \vee \neg P(Y,Z) \vee P(X,Z) \qquad P(a,b)}{\neg P(b,Z) \vee P(a,Z)}$$

- UR-resoluce

$$\frac{\neg P(X,Y) \vee \neg Q(X,Z) \vee R(Y,Z) \qquad P(a,b) \qquad \neg R(b,c)}{\neg Q(a,c)}$$

- hyper-resoluce

$$\frac{\neg P(X,Y) \vee \neg Q(X,Z) \vee R(Y,Z) \qquad P(a,b) \vee S(a,c) \qquad Q(a,c)}{R(b,c) \vee S(a,b)}$$

- paramodulace

$$\frac{A(0,X) = X \qquad B(A(X,1),A(0,1))}{B(A(X,1),1)}$$

Prover9 i Otter podporuje Unix-like systémy i Windows (díky Cygwinu).

2.2 Mace4

Mace4[3] je často používán ve dvojici s Proverem, ke kterému představuje doplněk. Zatímco Prover9 hledá důkaz daného tvrzení, Mace4 hledá konečné modely představující příklad dané teorie, v praxi se používá na hledání protipříkladů na dané tvrzení. Neboli hledá takové struktury, které splňují dané axiomy a negaci dokazovaného tvrzení. Pokud takovou strukturu najde, určitě je dokazovaná formule nedokazatelná.

Tento systém prohledává pouze konečné struktury (modely) s konečnou doménou, struktura je reprezentována tabulkou jednotlivých operací nad danou doménou (v Mace4 vždy prvky $0..n-1$ pro doménu velikosti n, z čistě teoretického hlediska to může být jakákoli n-prvková množina). Prohledávání standardně začíná s dvouprvkovou doménou a postupně přechází na větší až do limitu nastaveného parametrem udaným ve vstupním souboru.

Mace4 podporuje Unix-like systémy i Windows (díky Cygwinu).

Prover9 (a jeho předchůdce Otter) i Mace4 jsou vytvářeny stejným člověkem, Williamem McCunem z University of New Mexico v Albuquerque (předtím dlouhou dobu pracoval v Argonne National Laboratory). Oba jsou napsány v jazyku C a jsou distribuovány pod GNU GPL.

2.3 Waldmeister

Waldmeister[4] je dokazovací systém specializovaný na rovnosti, ve své kategorii *UEQ* v každoročním klání systémů automatického dokazování CASC[11] pravidelně posledních 10 let obsazuje první místo, tedy díky specializaci dosahuje skvělých výsledků. Základem funkce Waldmeisteru je Knuth-Bendixův algoritmus (ve formě, která nikdy neskončí neúspěchem, což ovšem neznamená, že skončí), proto pracuje pouze s rovnostmi. Přepisovací systém nemusí být v reálu konečný, s tím si samozřejmě Waldmeister nedovede poradit.

Waldmeister je v současnosti vyvíjen v *Max-Planck-Institut für Informatik* Thomasem Hillenbrandem a Berndem Löchnerem. Podporované operační systémy jsou Linux, Solaris, Mac OS X a Windows (skrze Cygwin).

2.4 Paradox

Podobně jako Mace4, z něhož Paradox[5] vychází, pracuje na principu hledání modelů teorií. Jeho funkce spočívá v převodu problému na SAT. Systém postupně vytváří instance teorie pro velikosti domény od M do N (M a N jsou uživatelem zadané dolní a horní meze).

Oproti Mace4 získává Paradox výhodu díky vhodně voleným heuristikám, které zmírňují některé nevýhody přístupu převodu problému do SAT. Nepříjemným problémem je například až exponenciální nárust počtu proměnných, k čemuž se ještě musí připočíst netriviální množství pomocných proměnných. Paradox používá metodu dělení[12] (orig. *splitting*), která z menšího počtu klauzulí s větším počtem proměnných vyrobí větší počet klauzulí s menším počtem proměnných. Narozdíl od ostatních systémů, které řeší dělení exponenciálním algoritmem, Paradox používá algoritmus běžící v polynomiálním čase[12]. Prostor instancí dále Paradox zmenšuje pomocí statické eliminace symetrie.

Jedinými nevýhodami Paradoxu jsou absence dokumentace a fakt, že jeho vývoj byl nejspíš ukončen (Jeho tvůrci se věnují novému projektu jménem Equinox).

Paradox je/byl vyvíjen Koen Claessenem a Niklasem Sörenssonem, v letech 2003-2006 získal prví příčku v soutěži dokazovacích systémů CASC[11] v divizi SAT/Models. Hlavní část programu je implementována ve funkcionálním jazyce Haskell, zatímco část řešící SAT (v tomto případě program *miniSAT*) je napsána v C++. Paradox je distribuován pod GNU GPL[9] verzí 2.

2.5 E

Systém E[6] je založen na tzv. superpozičním kalkulu pro logiku prvního řádu. Narozdíl od většiny jiných nástrojů pro automatické dokazování funguje E čistě na principu rovností, i problémy jiného druhu převádí do systému rovností. Ke zvýšení výkonu přispívají hlavně techniky *sdíleného přepisování termů* (dochází k několika substitucím naráz), indexování termů[10] a další techniky strojového učení.

Jako svůj vstup akceptuje jazyk LOP obohacený o speciální predikát rovnosti. LOP je velice podobný jazyku Prolog bez předdefinovaných operátorů a aritmetiky.

E je vyvíjen na Mnichovské univerzitě Stephanem Schultzem, je psán v jazyku C a podporuje prakticky všechny Unixy a Unix-like operační systémy.

2.6 Další systémy

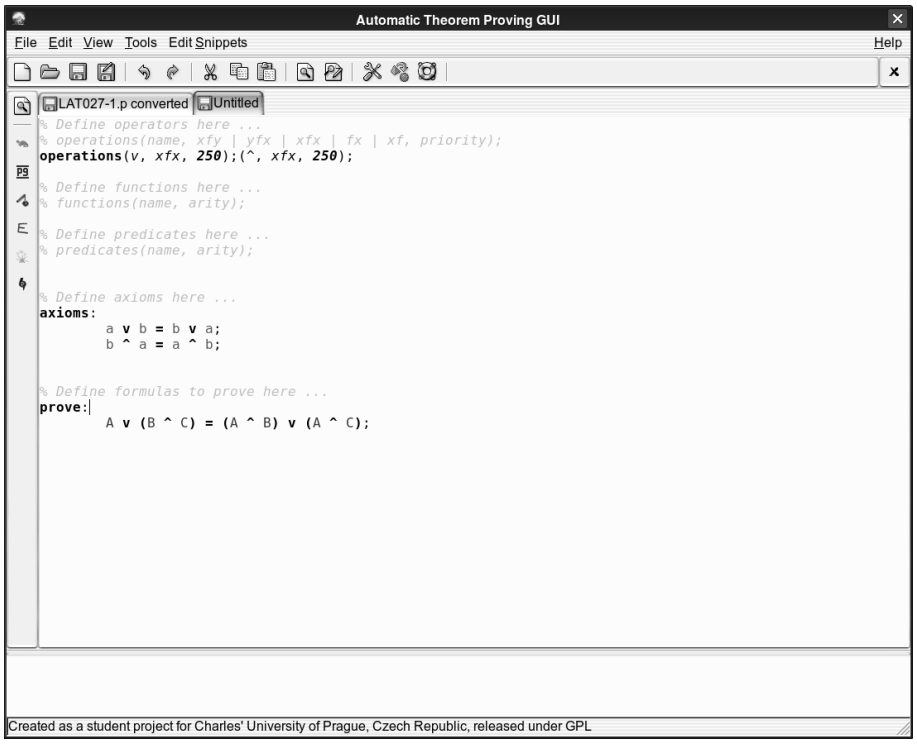
Během hledání informací o různých dokazovacích systémech jsem narazil na další systémy. I když jsem se jimi nijak nezabýval do hloubky, rád bych alespoň zmínil ty, které mě zaujaly.

V soutěžích dokazovacích systémů si v předešých letech velice dobře vedl *Vampire* z Manchesterské university od Andreje Voronkova a Alexandra Riazanova. Bohužel, nepodařilo se mi najít odkaz, který by nevedl na notoricky známou stránku *404 Not Found*. Z toho usuzuji, že vývoj Vampire už skončil. Bohužel, systém vznikal poněkud divoce, na webu nelze najít ani dokumentaci a popis algoritmu.

Z Velké Británie pochází též *Isabelle*. Tento systém je hlavně zajímavý tím, že se neomezuje na logiku prvního řádu, avšak nejedná se o automatický nástroj, nýbrž tzv. "proof assistant". Takovýchto systémů je vyvíjeno více (Coq, Theorema, HOL, Mizar, ...), ale v této práci se jim nebudu věnovat.

Zdaleka nejčastějším programovacím jazykem pro systémy automatického dokazování je C, protože velmi často je rychlost výpočtu zásadním kritériem úspěchu systému a také kompilátor C je dostupný na prakticky každé platformě, což je podstatná výhoda v momentě, kdy je třeba hledání důkazu provést na něčem jiném než osobním počítači. Najdou se samozřejmě výjimky, jako například *Paradox* (kombinace Haskellu a C++) nebo *Protein* v Prologu.

Automatické dokazování se taktéž neomezuje na samostatné počítače, například na McGill University se vyjíjí systém *Octopus*, který k dokazování používá paralelní běh na řádově stovkách počítačů.



2.7 TPTP a CASC

TPTP (Thousands of Problems for Theorem Provers[13]) je knihovna problémů pro automatické dokazování. Jejím hlavním účelem je porovnávání různých systémů. Tato knihovna obsahuje jak jednoduché problémy, tak problémy těžké i doposavad' otevřené.

Každý rok se koná v rámci *Conference on Automated Deduction* soutěž systémů *CADE ATP System Competition (CASC[11])*. Systémy automatického dokazování

na ní soutěží v řešení problémů z TPTP. Soutěžním kritériem v této soutěži není jen počet vyřešených úloh, ale i doba pořebná k řešení. Výsledky této soutěže měly velký vliv na výběr podporovaných systémů.

Největší devizou TPTP je právě knihovna problémů, která obsahuje celou škálu od jednoduchých po otevřené problémy. Mimoto poskytuje utilitu tptp2X, což není nic jiného než C-shell+Prolog skripty převádějící problémy z formátu TPTP do formátů přijímaných jednotlivými systémy. Největší nevýhodou tohoto formátu je ovšem jeho velká nepřehlednost. ATP GUI umožňuje otevírat soubory z TPTP (převede je do vlastního formátu). V CD příloze se též kromě samotné knihovny TPTP nachází plugin do ATP GUI umožňující převod z formátu ATP GUI do TPTP.

Kapitola 3

ATP GUI

V této části budu popisovat samotný program ATP GUI, jeho instalaci a ovládání. ATP GUI je grafické prostředí pro práci se systémy automatického dokazování matematických vět. Primárně je určen pro operační systém GNU/Linux, nicméně díky tomu, že používá knihovnu GTK+, lze jej používat i na jiných operačních systémech, například Windows. Podporováno je šest systémů automatického dokazování: Prover9[1], Otter[2], Mace4[3], Waldmeister[4], Paradox[5] a E[6].

3.1 Instalace v Linuxu

3.1.1 Samotná instalace

Instalační balíček ATP GUI pro Linux má podobu GZip archivu, prvním krokem v takovém případě zpravidla bývá jeho rozbalení. To se provede zadáním příkazu z adresáře, kde je stažený (či jinak získaný) balíček přítomen:

```
tar xvzf atpgui-0.1-linux.tar.gz
```

Tar a GZip obvykle bývají základní součástí linuxových distribucí. Po doběhnutí výše uvedeného se v aktuálním adresáři objeví adresář atpgui-0.1. Další kroky už předpokládají, že tento adresář je Váš aktuální adresář, neboli, že příkazová řádka bude vypadat nějak takto (v případě typického nastavení shellu):

```
...atpgui-0.1\$
```

Prvním krokem je otestování, jsou-li uspokojeny všechny závislosti. V krátkosti se jedná a knihovny Gtk+2.0 a libXML2, program pkg-config a samozřejmě kom-

pilátor jazyka C, zpravidla to bývá GNU C Compiler. K tomu slouží skript configure. K jeho vyvolání vepište do příkazové řádky

```
./configure
```

Tento skript zkontroluje nejen přítomnost potřebných součástí systému, ale také zajistí, že jejich verze bude dostatečná pro zkompilování ATP GUI. Pokud tomu tak není, upozorní na to příslušnou chybovou hláškou (popis a řešení dále). Takto dojde k nakonfigurování všeho potřebného tak, aby mohlo dojít k instalaci do adresářové struktury v `/usr/local`. Tato cesta je ve většině Linuxových distribucí standardní, avšak některé distribuce (například *ArchLinux*) tuto cestu ignorují při vyhledávání souborů ke spuštění (neboli `/usr/local/bin` není v PATHu), proto lze pomocí parametru `--prefix` skriptu `configure` definovat cestu jinou. Zejména to ocení lidé, kteří nemají práva zápisu do „systémových oblastí". Pokud bychom například chtěli nainstalovat program do adresáře `/usr/bin`, napsali bychom do příkazové řádky

```
./configure --prefix=/usr
```

Nutno ještě podotknout, že cesta musí být absolutní (neboli začínat lomítkem). Řekněme, že skript úspěšně skončil, potom se v aktuálním adresáři objevil soubor `Makefile`. Podle informací z tohoto souboru je kompilátor schopen vytvořit spustitelný soubor v nativním strojovém kódu.

Někdy může být žádoucí zkompilovat program bez podpory TPTP, například protože knihovna zabírá dost místa na disku, stačí jen spustit skript `configure` s parametrem `--omit-tptp`.

Po doběhnutí skriptu `configure` lze kompilaci vyvolat příkazem:

```
make
```

Kompilace zpravidla zabere delší dobu, po jejím úspěšném skončení se adresáři build nachází soubor `atpgui`. K dalšímu kroku typicky bývají nutná práva *root*a. V novějších distribucích bývá k dispozici nástroj *sudo*, lze použít i starší, léty prověřené *su*. Pokud ani jedna možnost nepřipadá v úvahu, je nutné změnit instalační prefix skriptu `configure`. Například takto:

```
./configure --prefix=~/atpgui-0.1
```

Tildu (∼) nahrazuje shell za domovský adresář, tedy tato cesta je absolutní.

V každém případě zbývá jen dokončit instalaci pomocí příkazu:

```
make install
```

Zbývá už jen dodat, že příkazem `make clean` lze smazat všechny soubory vytvořené příkazem make a příkazem `make distclean` ještě navíc soubory vytvořené skriptem `configure`.

3.1.2 Možné komplikace

Nejčastější komplikací nejspíše bude nepřítomnost programu *pkg-config* nebo jedné z knihoven *Gtk+* nebo *libXML2*. Zdrojové kódy ke všem těmto jsou sice volně ke stažení (všechny jsou licencovány GPL, nebo dokonce LGPL), avšak nedoporučoval bych snažit se je zkompilovat. Naprostá většina Linuxových distribucí disponuje nějakým balíčkovacím systémem (*apt-get*, *yum*, *pacman*, ...), pomocí kterého lze vše nainstalovat. Některé distribuce (například poslední dobou populární **Ubuntu*) mají ve zvyku instalovat pouze binární soubory knihoven bez hlavičkových souborů nutných ke kompilaci. Pro získání hlaviček je potom typicky třeba instalovat tzv. development balíček; typicky bývá označen příponou `-dev`. Bližší informace hledejte v dokumentaci Vaší distribuce.

Některé distribuce (a opět zmíním **Ubuntu*) mají taktéž podivný zvyk nezahrnovat kompilátor jazyka C do základní instalace. Bez kompilátoru toho mnoho nezkompilujete, proto je ho třeba nejdříve instalovat, opět Vás odkážu na dokumentaci Vaší distribuce.

3.1.3 Jiné unixové/Unix-like systémy

Výše uvedený postup se nijak neliší od běžného postupu kompilace na všech unixových a Unix-like systémech. Je možné, že ATP GUI půjde zkompilovat stejným způsobem i na dalších systémech, avšak **L**inux **i**s **n**ot **U**ni**x** a bohužel nemám možnost to rozumně otestovat.

3.1.4 Spuštění ATP GUI

Při standardní instalaci (tím se myslí instalace se standardním prefixem `/usr/local`, případně prefixem `/usr` apod.) se ATP GUI spustí příkazem

```
atpgui
```

Zda bude toto fungovat, záleží hlavně na nastavení proměnné prostředí (environment variable) PATH. Pokud shell tvrdí, že nemůže nalézt *atpgui*, je třeba mu dodat celou cestu k binárce, tj.

```
<PREFIX>/bin/atpgui
```

<PREFIX> nahrad'te hodnotou parametru `--prefix` skriptu `configure` nebo `/usr/local`, pokud nebyl specifikován. ATP GUI není třeba spouštět z příkazové řádky, grafická prostředí většinou umožňují vytvoření položky v menu nebo ikony na ploše, což je poněkud příjemnější cesta. Ke spuštění je též třeba běžící X server, pokud se nalézáte v grafickém prostředí, je tento předpoklad automaticky splněn.

3.2 Instalace ve Windows

3.2.1 Samotná instalace

ATP GUI pro Windows je Zip archiv s již zkompilovaným programem a všemi potřebnými knihovnami. Bohužel to též znamená, že je poněkud větší - 23 MB zabalený, 410 MB po rozbalení, velkou část z toho zabírá TPTP a soubory knihovny Gtk+. Podporovanou verzí Windows je jen Windows XP, starší verze, zejména Windows 9x, nejsou vhodné. Program není nutno speciálně instalovat, stačí jen rozbalit archiv `atpgui-0.1.3-win32.zip` a vše je připraveno pro spuštění. Rád bych ještě upozornil, že ATP GUI je primárně linuxový program, stejně jako naprostá většina systémů automatického dokazování. Ty většinou mohou být provozovány na Windows jen díky balíku Cygwin, což není nic jiného než emulace unixového/linuxového prostředí na Windows a některé systémy vůbec Windows ignorují. Taková emulace a vlastně i snaha přizpůsobit Windows něco, co není ve Windows běžné se samozřejmě projeví na výkonu. Problém spočívá hlavně ve skutečnosti, že systém Windows se soustřed'uje na jiné typy aplikací, než programy pracující v CLI, jak je typické pro unixové/unix-like prostředí.

3.2.2 Spuštění ATP GUI

ATP GUI se spouští dávkovým souborem `atpgui.bat`, případně, pokud je záhodno z jakéhokoli důvodu (rychlost) použít základní styl Gtk+, souborem `atpgui.exe`. Je nezbytné spouštět ATP GUI z pracovního adresáře stejného jako je lokace souboru `atpgui.exe`. Většinou to není problém, jelikož takto se Windows standardně chovají, je však třeba dávat si na toto pozor při vytváření zástupců.

3.3 Instalace rozšíření

3.3.1 Linux

Samotné ATP GUI neobsahuje žádné dokazovací systémy, ty je potřeba doinstalovat zvlášt'. V Linuxu se tak děje pomocí stejné sekvence příkazů jako instalace samotného programu. Naopak je nutné zadat stejný instalační prefix jako při instalaci samotného systému - skript `configure` kontroluje, je-li ATP GUI nainstalováno. Využijte tedy opět typické "svaté trojice"

```
configure
make
make install
```

Pro provedení třetího kroku budete nejspíš potřebovat rootovská oprávnění. Posledním krokem pro dokončení instalace je integrace systému do ATP GUI. Nejprve spust'te program ATP GUI a po vybrání v menu *Tools − > Import a new ATP system from XML* se zobrazí okno vyzývající Vás k výběru XML souboru s informacemi o systému automatického dokazování. Tyto soubory se nacházejí v adresáři `<PREFIX>/share/atpgui/systems`, při základním nastavení (těsně po instalaci) se otevře v nabídce právě tento adresář. Stačí tedy vybrat správný soubor a instalace je hotova.

3.3.2 Windows

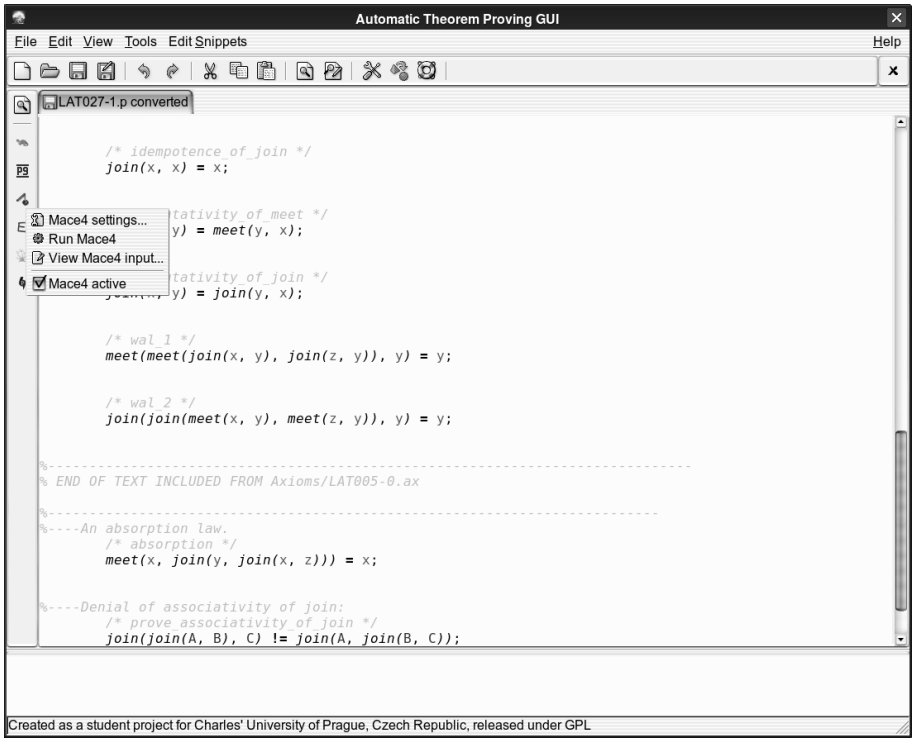
Pro instalaci zkopírujte archiv s rozšířením (*.zip soubor) do adresáře ATP GUI (do toho, kde se nalézá soubor `atpgui.exe`) a v tomto adresáři ho rozbalte. Je nutné archiv rozbalit tak, aby spustitelné soubory z tohoto archivu se po rozbalení ocitly ve stejném adresáři jako `atpgui.exe`. Původní archiv můžete smazat.

Pro dokončení instalace rozšiřujícího balíčku spust'te ATP GUI a po vybrání v menu *Tools − > Import a new ATP system from XML* se zobrazí okno vyzývající Vás k výběru XML souboru s informacemi o systému automatického dokazování. Tyto soubory se nacházejí v adresáři `systems` v adresáři ATP GUI, při základním nastavení (těsně po instalaci) se otevře v nabídce právě tento adresář. Stačí tedy vybrat správný soubor a instalace je hotova.

3.4 Práce se soubory

ATP GUI umožňuje díky tzv. „tabům" práci s několika soubory najednou, počet zároveň otevřených souborů je omezen jen velikostí paměti počítače a rozumem uživatele, který se chce v otevřených souborech orientovat. Všechny akce GUI (kromě Save all) se vztahují na soubor v právě aktivním „tabu".

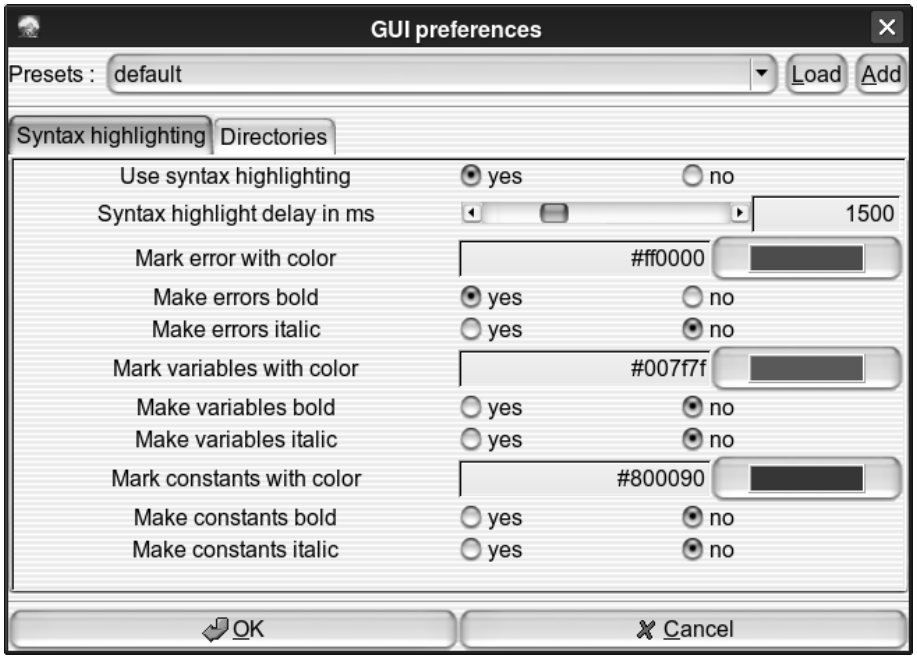
ATP GUI rozlišuje 2 základní typy souborů: Za prvé to je soubor specifikace problému; ten obsahuje definici axiomů a dokazovaných tvrzení. Druhým typem souborů jsou výsledky integrovaných systémů automatického dokazování; tyto jsou určeny jen pro čtení, tzn. ATP GUI neumožňuje je měnit. Soubor specifikace problému lze získat i ze souboru z knihovny TPTP. Slouží k tomu položka *Import TPTP file* v menu *File*.



Pro práci se soubory specifikace disponuje ATP GUI základní sadou nástrojů pro editaci textu, tj. tzv. „nekonečné undo/redo", schránka (Cut, copy a paste), vyhledávání textu, nahrazování textu a samozřejmě ukládání změn. Pro větší pohodlí a hlavně přehlednost disponují tyto soubory i možností zvýrazňování syntaxe. V základním nastavění je zvýrazňování zapnuté, avšak lze ho vypnout.

3.5 Nastavení

Nastavení ATP GUI se ukládá v souboru ∼/.atpgui_session (Linux) anebo v souboru atpgui.session (Windows). Tento soubor obsahuje nejen nastavení samotného programu, ale i jednotlivých integrovaných systémů. ATP GUI si díky tomu mezi jednotlivými spuštěními pamatuje všechna nastavení i otevřené soubory. Dojde-li však k poškození tohoto souboru, je třeba ho smazat a přijdete o všechna nastavení.



Viditelné součásti ATP GUI lze vypínat/zapínat pomocí jednotlivých položek v menu *View*. Takto lze vypnout levý i horní toolbar, zobrazování běžících procesů spuštěných ATP GUI a výběr „útržků definic" - snippets. Některé vlastnosti zvýrazňování syntaxe a nastavení adresářů lze měnit v nabídce zobrazení po aktivování položky menu *Tools* − > *GUI Preferences*.

3.6 Syntaxe

Soubor specifikace problému je seznam deklarací, kde jednotlivé deklarace jsou odděleny středníkem. Deklarace je uvozena klíčovým slovem určujícím její význam, avšak toto klíčové slovo lze vynechat. Potom se předpokládá, že deklarace je stejného typu jako deklarace předešlá. Každý funkční i predikátový symbol musí být defi-

nován dříve, než je použit. Symboly nelze přetěžovat (tzn. nelze například definovat f zároveň jako ternární funkci a binární predikát).

3.6.1 Operátory

Definice operátoru je uvozena klíčovým slovem **operations** a má následující tvar

```
operations(jmeno, typ, priorita);
```

jmeno značí jméno operátoru, může to být jakákoli sekvence znaků neobsahující závorky, středník, čárku, dvojtečku, mezeru, tabulátor nebo return (CR nebo LF). *typ* určuje aritu a asociativitu operátoru, možnosti jsou následující

- **fx** je prefixový unární operátor
- **xf** je postfixový unární operátor
- **xfx** je infixový binární operátor bez asociativity

  výraz $a + b + c$ je chybný

- **xfy** je infixový binární operátor s pravou asociativitou

  výraz $a + b + c$ je interpretován jako $a + (b + c)$

- **yfx** je infixový binární operátor s levou asociativitou

  výraz $a + b + c$ je interpretován jako $(a + b) + c$

Asociativita takto definovaná se týká pouze parsování, nikoliv skutečných vlastností daného operátoru, na toto je třeba dávat pozor. Priorita je celé číslo z rozmezí 1-1000, kde operátor s vyšší prioritou bude uplatněn první (například výraz a + b * c + d se při definicích (+,xfy,500) a (*,xfy,450) bude interpretovat jako a + ((b * c) + d) - + má vyšší prioritu a proto se nejdříve výraz rozdělí na dva podvýrazy podle +). Binární operátor musí být ve výrazu oddělen na obou stranách alespoň jednou mezerou, pokud začíná nebo končí písmenem, číslicí nebo podtržítkem.

3.6.2 Funkce

Definice funkce je uvozena klíčovým slovem **functions** a má následující tvar

```
functions(jmeno, arita);
```

Pro jméno funkce platí stejné omezení jako pro jméno operátoru. Při použití ve výrazu je nutno argumenty z obou stran obalit kulatými závorkami; jednotlivé argumenty jsou odděleny čárkou.

3.6.3 Predikáty

Definice predikátu je uvozena klíčovým slovem **predicates** a má následující tvar

```
predicates(jmeno, arita);
```

Pro jméno predikátu platí stále stejné omezení jako pro jméno operátoru nebo funkce. Při použití ve výrazu je nutno argumenty z obou stran obalit kulatými závorkami; jednotlivé argumenty jsou odděleny čárkou.

3.6.4 Proměnné a konstanty

Proměnné a konstanty není třeba explicitně definovat předem, neznámé identifikátory budou interpretovány bud' jako proměnné (začínají-li na malé písmeno anglické abecedy - tj. [a-z]), nebo konstanta (pokud nemuže být proměnná).

Mezi konstantami mají zvláštní postavení číselné konstanty - pro systémy hledající modely značí zcela konkrétní prvek domény ($\Rightarrow$ pokud mám konstanty A a B, může se stát že $A = B$, ale nestane se, že $5 = 7$). Navíc, výskyt takovéto konstanty znemožňuje vytvoření menšího modelu než hodnota konstanty+1.

3.6.5 Axiomy

Definice axiomu je uvozena klíčovým slovem **axioms :** (před dvojtečkou může být libovolný počet mezer, tabulárotů a new-lineů). Za ním následuje samotný axiom (jakožto formule).

3.6.6 Dokazovaná tvrzení

Definice dokazovaného tvrzení je uvozena klíčovým slovem **prove :** (před dvojtečkou může být opět libovolný počet mezer, tabulárotů a new-lineů). Za ním následuje samotné tvrzení (jakožto formule).

3.6.7 Uspořádání na termech

Většina systémů umožňuje definovat uspoření na termech, nejčastěji *Lexicographic Path Ordering* - LPO nebo Knuth-Bendix Ordering - KBO. Nastavení těchto uspořádání může výrazně ovlivnit výpočet. Základní (v tomto případě lexikografické) řazení se deklaruje klíčovým slovem *order* a za ním, v kulatých závorkách vzestupný seznam symbolů (operátory, funkce, predikáty, proměnné, konstanty), tedy

```
order(+, -, 0);
```

odpovídá řazení $+ < - < 0$. Takováto definice se může v souboru vyskytnout jen jednou, další výskyty budou označeny jako chybné. Knuth-Bendix ordering navíc počítá s ohodnocením symbolů pomocí klíčového slova *weight*. Za ním v závorce následuje dvojice složená z názvu symbolu a jeho váhy (celé nezáporné číslo). Například

```
weight(+,3);
```

přiřazuje symbolu + pro KBO váhu 3. V souboru specifikace problému nemusí být vůbec řazení termů definováno, potom se předpokládá, že systém nějaké řazení vyrobí automaticky, stejně tak nemusí být třeba vždy definovat řazení celé, někdy je možné spoléhat se na to, že systém sám zbylé symboly zařadí.

3.6.8 Předdefinované symboly

ATP GUI definuje některé symboly automaticky, předně to jsou logické spojky jako binární operátory `->`, `<->` (bez asociativity), &, | (s pravou asociativitou), potom jako unární operaci ! jakožto negaci, dále predikáty rovnosti a nerovnosti, avšak jako binární operátory (kvůli infixu) `=` a `!=` (bez asociativity). Dále jsou definovány operátory +(priorita 450) a ∗(priorita 400); oba s pravou asociativitou, unární prefixový operátor − a postfixový ′. V neposlední řadě jsou definovány oba kvantifikátory, i když některé systémy si s nimi neumí poradit.

```
(Vx)(Ey)(x + y > x)
```

univerzálně kvantifikuje x a existenčně y

3.6.9 Příklad souboru specifikace

Příklad specifikace problému by mohl vypadat například takto:

```
operations(v,xfy,150);
functions (i,1);
axioms:
     a + b = b + a;
     (a + b) + c = a + (b + c);
     a * b = b * a;
     (a * b) * c = a * (b * c);

     0 + a = a;
     1 * a = a;
     0 * a = 0;
     a * (b + c) = (a * b) + (a * c);
     -a + a = 0;

     a v b = b v a;

     (Vz)((!(z = 0)) -> (i(z) * z = 1));
     (Vz)((i(z) * z = 1) -> (!(z = 0)));

prove : 1 != 0;

order(+, *, 1, 0, -, i, v);weight(+,1);(*,1);(1,2);
```

3.7 Zvýrazňování syntaxe

Syntaxe ATP GUI je bezkontextová gramatika, avšak parsovat ji jako obecnou bezkontextovou gramatiku zde není možné kvůli rychlosti. Původně jsem se o to pokoušel, ale standardní lexikální a syntaktická analýza zabraly příliš mnoho času. Proto jsem musel přistoupit k méně obecnému způsobu. Struktura souboru je reprezentována pomocí stromu (struktura `parse_tree`) s označením začátku a konce části textu, kterého se strom týká. Text je strukturován pomocí následujících pravidel: (neterminální symboly jsou vyznačeny dvojicí špičatých závorek, počáteční terminál je <SOUBOR>, λ značí prázdné slovo).

$$
\begin{array}{lcl}
< SOUBOR > & \rightarrow & < PRIKAZ >; < SOUBOR > \\
 & & \lambda \\
< PRIKAZ > & \rightarrow & operations(< JMENO >, < ASOC >, < CISLO >) \\
 & & functions(< JMENO >, < CISLO >) \\
 & & predicates(< JMENO >, < CISLO >) \\
 & & weight(< JMENO >, < CISLO >) \\
 & & order(< JMENO >< CONT >) \\
 & & axioms :< VYRAZ > \\
 & & prove :< VYRAZ > \\
< JMENO > & \rightarrow & \neg[\ \backslash t \backslash n, ;] < JMENOL > \\
< JMENOL > & \rightarrow & \neg[\ \backslash t \backslash n, ;] < JMENOL > \\
 & & \lambda \\
< CISLO > & \rightarrow & [0123456789] < CISLOL > \\
< CISLOL > & \rightarrow & [0123456789] < CISLOL > \\
 & & \lambda \\
< ASOC > & \rightarrow & xfx \mid xfy \mid yfx \mid fx \mid xf \\
< CONT > & \rightarrow & , < JMENO >< CONT > \\
 & & \lambda
\end{array}
$$

Okolo speciálních znaků (čárka, tečka, závorky) lze vkládat libovolný počet mezer, tabulátorů nebo new-lineů. Neterminál <`VYRAZ`> jsem byl kvůli rychlosti (ta ani ted' není bůhvíjaká) parsovat trošku proti pravidlům bezkontextových gramatik, bohužel u nich není ekvivalentní deterministická a nedeterministická forma a nedeterminismus všeobecně v dnešních počítačích činí potíže. Výrazy se proto parsují takto:

- Nejdříve se zkontroluje, že výraz je správně uzávorkován (pokud není, označí se celý výraz za chybný).
- Dále se hledají binární operátory od toho s největší prioritou k tomu s nejmenší. Hledá se jen v částech výrazu, které nejsou uzávorkované. Pokud se najde více výskytů operátoru, vybere se jeden na základě typu jeho asociativity - u levé asociativity se vybere nejpozdější výskyt, u pravé nejdřívější. Pokud nemá operátor žádnou asociativitu, označí se výraz jako chybný. Části výrazu nalevo i napravo od operátoru se rekurzivvně zpracují stejným způsobem.
- Není-li nalezen binární operátor, hledá se unární - prefixový se hledá na začátku výrazu, postfixový na konci.
- Dále se hledá funkční nebo predikátový symbol na začátku výrazu, musí však být následován uzávorkovaným výčtem argumentů oddělených čárkami. Pokud

není, je výraz označen za chybný. Argumenty jsou rekurzivně zpracovány jako výrazy.

- Dále se pokouší parser brát výraz jako kvantifikovanou formuli $(Vx)...$ nebo $(Ex)....$ Kvantifikátory se hledají až po binárních operátorech, tedy vážou méně, proto je výhodné (rozhodně méně matoucí) označit kvantifikovanou část formule závorkami, například $(Vx)(p(x) -> q(x))$.

- Pokud ani to nelze, pokusí se parser vzít výraz jako uzávorkovaný, pokud začíná i končí závorkou, vnitřek závorky je rekurzivně zpracován jako výraz.

- Poslední možností je, že výraz je proměnná nebo konstanta, pokud se výraz nekvalifikuje jako proměnná nebo konstanta (například protože uprostřed je mezera), je označen výraz za chybný.

Výsledkem je strom, jehož listy jsou proměnné, konstanty a chybné části výrazu. U každé ho uzlu je poznamenáno o jaký typ uzlu se jedná (jestli je to konstanta nebo binární operátor rozdělující výraz na dvě části atd.) a podle toho je příslušná část zabarvena, ztučněna, zkosena etc. Do textu lze různě vkládat mezery, tabulatory, odřádkovaní nebo komentáře (ovšem ne doprostřed identifikátorů nebo klíčových slov). Ke zvýrazňování syntaxe se používá tentýž postup jako při předzpracovávání vstupu pro jednotlivé systémy, tedy je zajištěno, že vstup bude správný, jestliže uživateli nebude žádná část jeho definic označena jako chybná.

3.8 Snippets

Snippets jsou útržky definic, které lze vkládat do souboru specifikace problému. Typicky je vhodné je použít na uložení často používané sady operátorů a axiomů (například člověk pracující se svazy si může uložit definice spojení a průseku včetně axiomů). Snippets mohou být rozděleny do několika skupin. V základním nastavení nejsou žádné snippets ani skupiny vytvořeny. Skupinu lze přidat v menu *Edit snippets* $->$ *Add a new group*. Po vyplnění a potvrzení formuláře vznikne nová skupina a v menu vznikne nová položka s názvem skupiny. Pomocí tohoto submenu lze vkládat do skupiny snippety, změnit nastavení skupiny nebo ji kompletně smazat. Přidání nového snippetu se děje přes podobný dialog jako přidání skupiny. Kliknutím na název snippetu v menu skupiny můžete měnit jeho obsah.

Takto vytvořené části definic můžete potom vkládat do textu pomocí toolbaru umístěného pod samotným textem souboru (pokud tam nic není, je pravděpodobně toolbar snippets schován a lze ho zobrazit zaškrtnutím položky *Snippets* v menu

View). Po kliknutí na ikonu skupiny se zobrazí kontextové menu, ve kterém si můžete vybrat konkrétní snippet ke vložení.

3.9 Systémy automatického dokazování

Nainstalované systémy automatického dokazování se zobrazují jako ikony v levé části okna aplikace. Při kliknutí na ikonu daného systému se zobrazí kontextové menu s následujícími možnostmi:

- **Settings**

  Pod touto položkou menu se skrývá nastavení parametrů hledání důkazu. Tato nastavení jsou specifická pro každý systém. Pro většinu nastavení lze vyvolat krátkou nápovědu při podržení kurzoru myši nad textovým polem nebo zaškrtávátky jednotlivého parametru.

  Nastavení mohou významným způsobem ovlivnit průběh výpočtu a často se může stát, že pro některé problémy je nějaké nastavení lepší než jiné. Pro takový případ se hodí možnost uložit si několik různých nastavení a poté mezi nimi přepínat. K tomu slouží *Combo-box* na vrchu okna s nastavením a tlačítka *Add* a *Load*.

- **Run**

  Touto položkou menu započnete hledání důkazu daným systémem automatického dokazování. Obsahuje-li specifikace problému více formulí pod klíčovým slovem *prove :*, bude potom pro každou vytvořena vlastní instance dokazovacího systému. Běžící procesy spuštěné ATP GUI se zobrazují ve spodní části okna (pokud není daný prvek explicitně skryt pomocí menu *View*). Pokud nějaký proces běží příliš dlouho, lze ho ukončit kliknutím na jeho ikonu pravým tlačítkem myši a vybráním položky *Stop* zobrazeného kontextového menu. V tomtéž menu si taktéž lze pomocí položky *Watch* zobrazit okno s částečným výstupem programu.

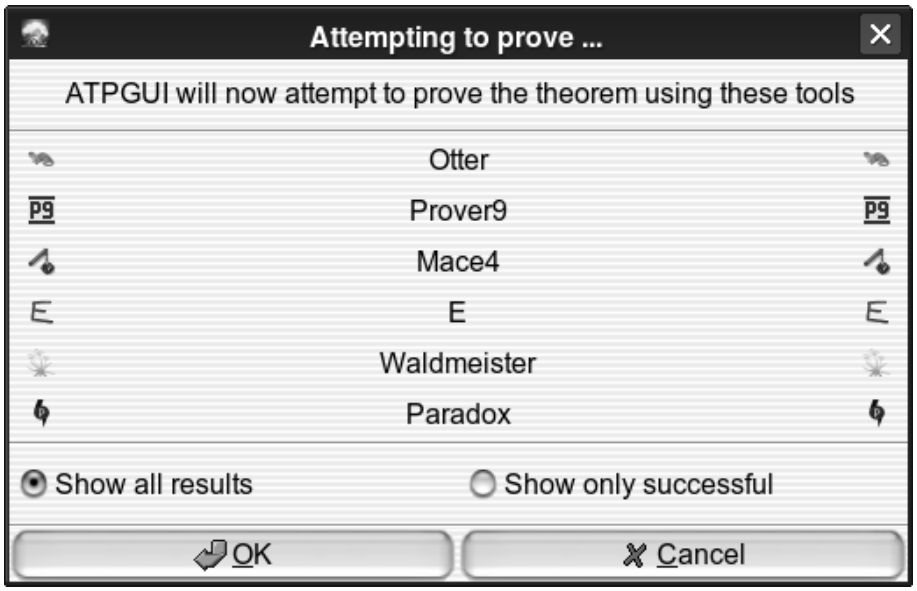
- **View input**

  Zobrazí v hlavním okně předzpracovaný vstup pro daný systém.

- **Active**

Podle stavu zaškrtnutí toho políčka se určuje, bude-li systém zahrnut mezi systémy použité při paralelním dokazování několika systémy při stisku tlačítka Prove.

Nad ikonkami systémů se nachází též ikonka s popiskem Prove. Kliknutím na ni spustíte všechny systémy, které mají zašktnutou položku active. Hlavní rozdíl tohoto přístupu proti spuštění jednotlivých systémů samostatně spočívá v tom, že takto se v okně mohou nechat v ATP GUI zobrazit jen výstupy úspěšně ukončených systémů (tzn. byl nalezen důkaz nebo protipříklad).



3.10 Plug-iny

V menu *Tools* se mj. vyskytují i další volitelné nástroje pro konverzi vstupního textu (například konverze do formátu *TPTP*). Tyto nástroje mají podobu dynamicky načítaných modulů - v Linuxu to jsou knihovny s příponou `.so` (shared object) a ve Windows knihovny s příponou `.dll` (Dynamically linked library). Seznam používaných modulů se nachází v souboru *plugins.list*, kde je pro každý modul vyhrazena jedna řádka. Tyto moduly fungují tak, že načtou vstupní soubor napsaný uživatelem a do nového „tabu" otevře výsledek práce modulu. Díky tomu, že se jedná o běžné sdílené knihovny, mají případní autoři knihoven možnost například zobrazit dialogové okno s formulářem dotazujícím se na další informace od uživatele.

3.11 Vnitřní fungování programu

ATP GUI je napsáno s pomocí knihovny GTK+ a nijak zásadně se neliší od ostatních programů vytvořených pomocí této knihovny. Základem programu je struktura `atpgui_settings_struct`, která se vyskytuje v paměti jen jednou. Celý program je do značné míry vizualizací dat v této struktuře a změn v ní. V souboru `.atpgui_session` (`atpgui.session`) je uložena právě tato struktura, program ji po spuštění načte (pokud se to z nějakého důvodu nepodaří, vytvoří vlastní, obsahující jen základní data). Podle této struktury se vytvoří widgety Gtk+ a dál už jen program běží ve smyčce a zpracovává události, dokud nedostane povel k ukončení. Na to program zareaguje pokusem o uložení rozdělané práce a uložení výše zmíněné struktury, načež ukončí svou činnost.

Veškeré změny ve struktuře `atpgui_settings_struct` jsou prováděny pomocí tzv. callbacků přiřazených k jednotlivým událostem. V programu jsem důsledně zachovával lokalitu, tedy nedeklaroval jsem jedinou globální proměnnou.

`Atpgui_settings_struct` obsahuje další podstruktury. Předně je to dvojitý spojový seznam otevřených souborů (což je opět struktura), dvojitý spojový seznam systémů (to je opět struktura obsahující dvojitý spojový seznam vlastností systému), strukturu s procesy a jejich I/O buffery uloženými v dynamicky se rozšiřujících polích, strukturu se skupinami *snippets* a *snippets* samotnými, strukturu se strukturami reprezentujícími plug-iny uloženými opět v dynamicky se rozšířujícím poli a nakonec strukturu obsahující vlastnosti samotného GUI ve dvojitém spojovém seznamu. Ačkoli C je samo o sobě neobjektový jazyk, kód je hodně objektový.

Kapitola 4

Specifikace rozšíření o další systémy

Tato kapitola obsahuje (nebo by alespoň měla) všechny potřebné informace pro vytvoření rozšíření integrovatelného do ATP GUI. Program byl od začátku navrhován tak, aby vytvoření takovéhoto balíčku bylo co nejjednodušší, aby šlo snadno přidávat nové verze systémů i nové systémy.

4.1 Soubory

Rozšíření se hlavně skládá z několika malých programů, které přečtou data ze standardního vstupu, provedou nějaké operace a vypíší transformovaná data na výstup, kde na ně čeká další program, aby je mohl načíst atd. V balíčku nutně musí být program, který převede parametry výpočtu do formátu daného systému, program, který převede definice z formátu ATP GUI do formátu daného systému, program samotného systému a nakonec program, který zpracuje výstup systému a pošle ho zpět ATP GUI. V praxi to funguje tak, že výstupy prvních dvou programů se spojí a pošlou na standardní vstup systému, ten se potom pokouší dokázat nebo vyvrátit dané tvrzení a jeho výstup se pošle čtvrtému programu. Aby program ATP GUI věděl, jaké programy k danému účelu spouštět, musí být součástí balíku i XML soubor, ze kterého ATP GUI vyčte potřebné informace. Kromě povinných součástí se může v balíku nalézat i program konvertující výstup systému do LaTeXu (avšak žádny z balíčků v příloze této práce toho nevyužívá). Další nepovinnou částí je ikonka daného systému ve formátu PNG, nejlépe o rozměrech 16x16 pixelů. Navíc některé systémy neumí číst data ze standardního vstupu, potřebují soubor na disku (to se týká jen Windows, na čemkoli alespoř trochu podobném Unixu stačí jako vstupní soubor v takovém případě poslouží `/dev/stdin`), a proto u takových systémů se v

balíčku vyskytne wrapper (typicky bash skript - díky Cygwinu ho lze použít i na Windows), aby se systém choval podle představ ATP GUI.

Každý proces spuštěný v ATP GUI je vytvořen pomocí funkce knihovny Glib `g_spawn_async_with_pipes`. Na `stdin` dostane data připravená ATP GUI. O čtení ze `stdout` se stará speciální vlákno (používat více vláken by nebylo nutné, pokud by Glib pro Win32 uměla neblokovací čtení z roury), které jen po blocích čte výstup spuštěného procesu. Informace o běžících procesech a vstupní/výstupní buffery jsou spravovány ve struktuře `atpgui_processes`.

V dalších částech se budu věnovat jedlotlivým součástem, které je třeba vytvořit. Programy u rozšíření, která jsou přílohou této práce, jsou programovány v C nebo Bashi, nicméně výběr programovacího nebo skriptovacího jazkyka je samozřejmě neomezen.

4.2 XML definice rozšíření

4.2.1 <atp-system>

Kořenovým elementem souboru definice rozšíření je `atp-system`. Ten může mít dva atributy

- **name** - Určuje jméno systému. Pokud už bude v ATP GUI integrován systém stejného jména, bude tento nahrazen systémem z tohoto XML souboru. Proto je nutné volit unikátní jméno pro každý systém. Pokud by se počítalo se souběžným používáním několika verzí téhož systému, je dobré v názvu uvádět i verzi. Tento atribut je povinný.
- **icon** - Určuje jméno souboru s ikonou, jedná se o obrázek PNG s preferovanými rozměry 16x16 pixelů. Lze použít i jiné rozměry, ale v ATP GUI bude zobrazena v originální velikosti, takže při neuvážené velikosti může dojít k narušení vzhledu programu.

Vlastnosti tohoto elementu jsou dále specifikovány dalšími subelementy.

4.2.2 <gui2system>

Obsahem tohoto elementu je příkazová řádka ke spuštění *konvertoru parametrů výpočtu*. Řádka bude použita přesně v tomto znění, proměnné prostředí nebudou

nahrazeny jejich hodnotami. Není nutné zadávat celou cestu k programu, pokud se nachází v adresáři obsaženém v proměnné prostředí (cesty v `PATH`u budou prohledány), avšak doporučuje se uvádět celou cestu, pokud je to možné. Pokud se vyskytne v souboru tento element vícekrát, bere se v úvahu jen poslední výskyt.

4.2.3 <prop2system>

Obsahem tohoto elementu je příkazová řádka ke spuštění *konvertoru vstupu*. Platí zde stejná pravidla jako u předešlého elementu.

4.2.4 <system>

Obsahem tohoto elementu je příkazová řádka ke spuštění samotného systému. Platí zde stejná pravidla jako u elementu gui2system.

4.2.5 <postprocessing>

Obsahem tohoto elementu je příkazová řádka ke spuštění *konvertoru výstupu*. Platí zde stejná pravidla jako u elementu gui2system.

4.2.6 <system2tex>

Obsahem tohoto elementu je příkazová řádka ke spuštění *konvertoru do LaTeXu*. Platí zde stejná pravidla jako u elementu gui2system.

4.2.7 <group>

Element *group* představuje skupinu nastavitelných vlastností hledání důkazu nebo systému jako takového. Všechny vlastnosti z této skupiny potom bude v ATP GUI ve vlastnostech integrovaného systému zobrazena v jednom „tabu". Tento element má povinný atribut *name*, jehož hodnota navíc musí být mezi jmény skupin unikátní. Obsahem tohoto elementu je seznam elementů *property*.

4.2.8 <property>

Elementy *property* definují samotné nastavitelné vlastnosti hledání důkazu a/nebo vlastnosti samotného systému. Obsahem elementu je jméno vlastnosti a, stejně jako

všechna jména, i toto jméno musí být v rámci jmen vlastností unikátní. Element *property* rozaznává následující atributy:

- **type** - Udává obor hodnot pro danou vlastnost. Možnosti jsou následující
  - **int** - Hodnoty vlastnosti jsou celá čísla, pomocí atributů *min*, *max* je třeba definovat rozmezí, ve kterém se budou hodnoty pohybovat. V ATP GUI bude takováto vlastnost zobrazena jako scrollbar a textové pole.
  - **boolean** - Hodnoty vlastnosti jsou bud' zapnuto (*true*), nebo vypnuto (*false*). V ATP GUI bude zobrazeno jako dvojice radio-buttonů (s popisky *yes* a *no*).
  - **string** - Hodnotami jsou řetězce znaků. Bude zobrazeno jako textové pole.
- **default** - Udává základní hodnotu parametru.
- **min**, **max** - Mají smysl jen pro vlastnost typu *int*, pro kterou udávají minimální, resp. maximální hodnotu.
- **hint** - Text kontextové nápovědy, která se zobrazuje při delším podržení kurzoru nad nastavením parametru v ATP GUI.

4.2.9 Příklad XML definice systému

```
<atp-system name="Waldmeister" icon="/usr/share/icons/atpgui/w.png">
<gui2system>/usr/bin/waldmeister_convert</gui2system>
<prop2system>cat</prop2system>
<system>/usr/bin/waldmeister_wrapper /usr/bin/waldmeister</system>
<system2tex></system2tex>
<postprocessing>/usr/bin/waldmeister_postprocessing</postprocessing>
<group name="Properties">
<property type="boolean" default="false" hint="If this flag is set,
  the automated control component, including  construction
  of a reduction ordering and selection of a weighting function
  is activated. This also means that the &quot;Use add weight
  heuristics&quot; flag has no effect on the proof search.
  ">Use auto mode</property>
<property type="boolean" default="false" hint="If this flag is set,
  add-weight heuristic is used (term size measuring), otherwise
  the mix-weight heuristic is used.">Use add weight heuristics
  </property>
```

```
</group>
</atp-system>
```

4.3 Konvertor parametrů výpočtu

Konvertor parametrů výpočtu dostane na standarní vstup (`stdin`) seznam všech vlastností, kde každá vlastnost je reprezentována dvěma řádkami, na první řádce je jméno vlastnosti a na druhé hodnota. Úkolem tohoto programu je potom transformovat vstup do takové podoby, která bude čitelná pro cílový systém a vypsat výsledek na standardní výstup (`stdout`).

Některé systémy automatického dokazování nepřijímají parametry výpočtu ve vstupním souboru, nýbrž jako parametry na příkazové řádce. V takovém případě lze jako Konvertor parametrů výpočtu použít program `cat`, který vypíše vstup nepozměněný na výstup a okolo systému naprogramovat tzv. wrapper, což je většinou Bash-script (ale může to být i jiný skript nebo program), který spustí systém s potřebnými parametry.

4.4 Konvertor vstupu

Konvertor vstupu dostane na standardní vstup (*stdin*) předzpracovaný soubor specifikace problému. Předzpracovaný soubor obsahuje axiomy a dokazovaná tvrzení v čistě prefixové formě. Všechny operátory, funkce, predikáty, proměnné i funkce jsou přejmenovány (kromě operátorů =, `!=`, `!`, `->`, |, &, `<->`). Důvodem tohoto obratu je skutečnost, že některé systémy některým běžným operátorům přisuzují zvláštní vlastnosti (např. komutativita +), zatímco jiné ne. Kvantifikátory jsou převedeny do podoby `forall(proměnná,formule)` pro univerzální kvantifikátor a `exists(proměnná,formule)` pro existenční. Vstup programu má následující strukturu

- Řádka

  `OPERATORS START`

  značí začátek definice operátorů, následující řádky budou považovány za definice operátorů, dokud se nenarazí na řádku s obsahem

  `OPERATORS END`

Mezi tím jsou definovány všechny operátory. Definice operátoru se skládá z následujících 5 řádek

  - **OPERATOR**
  - **nové jméno operátoru** - toto jméno je pro operátor používáno ve vstupní definici axiomů a dokazovaných tvrzení.
  - **původní jméno operátoru** - jméno, které používá uživatel ATP GUI.
  - **arita** - *BINARY* nebo *UNARY*.
  - **asociativita / pozice** - U binárních operátorů *NOT-ASSOCIATIVE*, *LEFT-ASSOCIATIVE* nebo *RIGHT-ASSOCIATIVE*; u unárních *PREFIX* nebo *POSTFIX*.
  - **váha pro KBO** - pokud není definována, potom se uvádí -1.

- Řádka

  `FUNCTIONS START`

  značí začátek definice funkcí a predikátů, následující řádky budou považovány za definice funkcí a predikátů, dokud se nenarazí na řádku s obsahem

  `FUNCTIONS END`

  Mezi tím jsou definovány všechny funkce a predikáty. Definice se skládá z následujících 4 řádek

  - **FUNCTION**
  - **nové jméno funkce/predikátu** - toto jméno je pro funkci/predikát používáno ve vstupní definici axiomů a dokazovaných tvrzení.
  - **původní jméno funkce/predikátu** - jméno, které používá uživatel ATP GUI.
  - **arita** - Celé číslo > 0.
  - **váha pro KBO** - pokud není definována, potom se uvádí -1.

- Řádka

  `CONSTANTS START`

  značí začátek deklarace konstant, následující řádky budou považovány za deklarace konstant, dokud se nenarazí na řádku s obsahem

`CONSTANTS END`

Mezi tím jsou deklarovány všechny konstanty. Definice se skládá z následujících 2 řádek

- **nové jméno konstanty** - toto jméno je pro konstantu používáno ve vstupní definici axiomů a dokazovaných tvrzení.
- **původní jméno konstanty** - jméno, které používá uživatel ATP GUI.
- **váha pro KBO** - pokud není definována, potom se uvádí -1.

- Řádka

`VARIABLES START`

značí začátek deklarace proměnných, následující řádky budou považovány za deklarace proměnných, dokud se nenarazí na řádku s obsahem

`VARIABLES END`

Mezi tím jsou deklarovány všechny proměnné. Definice se skládá z následujících 2 řádek

- **nové jméno proměnné** - toto jméno je pro proměnnou používáno ve vstupní definici axiomů a dokazovaných tvrzení.
- **původní jméno proměnné** - jméno, které používá uživatel ATP GUI.
- **váha pro KBO** - pokud není definována, potom se uvádí -1.

- Řádka

`ORDER START`

Značí začátek (vzestupně setřízeného) seznamu symbolů s definovanou precedencí pro lexikální třídění. Konvertor nemusí nijak doplňovat tento seznam, pokud není kompletní - je již na samotném systému, jak se s touto situací vypořádá. Seznam je ukončen řádkem

`ORDER END`

- Řádka

`AXIOM`

značí, že na další řádce se nachází definice axiomu.

- Řádka

  `PROVE`

  značí, že na další řádce se nachází definice dokazovaného tvrzení.

Ostatní řádky jsou považovány za komentáře. Úkolem konvertoru vstupu je tato data načíst a na standardní výstup vypsat axiomy a jedno dokazované tvrzení ve formátu srozumitelném danému systému a ukončit soubor řádkou

```
#end_of_file
```

Pokud se ve vstupu nachází více jak jedno návěstí `PROVE`, musí pro každé dokazované tvrzení vypsat vlastní soubor, konec jednoho souboru a začátek dalšího se pozná právě podle výše zmíněné řádky. Předzpracované formule většinou už není třeba dále upravovat, prefixovou formu akceptují dokazovací systémy spíše než infixovou, nicméně někdy může být třeba formule upravovat, například Paradox[5] vyžaduje vstup v CNF.

Samotný systém potom dostane na vstup výstup konvertoru parametrů výpočtu a za ním hned výstup konvertoru vstupu.

Pokud výstup systému začíná řetězcem `OVERRIDE`, znamená to, že ATP GUI nespustí rovnou konvertor výstupu, nýbrž příkaz, který se nachází hned (žádná mezera) za tímto slovem, a teprve na výstup nového programu se použije konvertor výstupu. Ve verzi pro Windows takto fungují Waldmeister a Paradox.

4.5 Konvertor výstupu

Konvertor výstupu dostane na vstup ta samá data jako konvertor vstupu následovaná řádkou

```
#end of part 1
```

a potom následuje výstup systému. Úkolem tohoto programu je převést zpět formule do infixového tvaru a přejmenovat identifikátory na jejich původní jména. Druhým úkolem je z výstupu systému zjistit, jestli byl nalezen důkaz nebo protipříklad. V takovém případě na první řádek výstupu vypíše:

```
#success
```

V opačném případě vypíše:

```
#failure
```

Za udáním úspěšnosti následuje zpracovaný výstup systému.

4.6 Konvertor do Latexu

Konvertor do Latexu dostane na vstup výstup předešlého programu a jeho úkolem je ho převést do Latexu. Narozdíl od předchozích programů se jedná o nepovinnou součást balíčku.

4.7 Struktura adresářů

Struktura adresářů se liší v závislosti na platformě.

4.7.1 Linux

Adresář s rozšířením je archivován jako GZip nebo BZip(2) archiv. V kořenovém adresáři archivu je jen adresář s názvem `atpgui_<jméno systému>`. V tomto adresáři se dále nachází

- skript `configure`. Ten je zodpovědný za vytvoření souboru `Makefile`, dá se velice dobře použít upravený skript z některého ze stávajících balíků.
- textový soubor `AUTHORS` obsahující jména autora nebo autorů
- textový soubor `INSTALL` obsahující instrukce k instalaci rozšíření
- textový soubor `COPYING` obsahující text licence GPL, tedy pokud je rozšíření licencováno pod GPL
- adresář `build` sloužící k ukládání výsledků kompilace a linkování.
- adresář `icons` obsahující ikonu systému, pokud nějaká je.
- adresář `import` obsahující zdrojový text pro skript `configure`, aby z něj vyrobil XML definici systému.

Další součásti rozšíření ATP GUI již jsou specifické pro daný systém a je již věcí autora, jak rozšíří tuto adresářvou strukturu, nicméně je důležité, aby k instalaci stačila trojice příkazů `./configure` && `make` && `make install`, a je opět záležitostí autora, jak toho dosáhnout.

4.7.2 Windows

Adresář s rozšířením je archivován jako Zip archiv, může samozřejmě jít o jiný formát archivace souborů, není to až tak podstatné. V kořenovém adresáři se nalézají tyto soubory a podadresáře:

- textový soubor `AUTHORS` obsahující jméno autora/jména autorů.
- textový soubor `COPYING` obsahující text GNU GPL, pokud tedy balíček je licencován pod touto licencí.
- textový soubor `<jméno>_plugin_README.txt` obsahující instrukce pro instalaci balíčku.
- adresář `icons` obsahující ikonu systému.
- adresář `systems` obsahující XML definici systému
- adresář `srcs` obsahující zdrojové kódy jednotlivých součástí.

Další součásti už jsou specifické pro daný systém. Tvůrce balíčku může automaticky předpokládat, že ve Windows bude ATP GUI a tedy i jeho programy spouštěny s pracovním adresářem shodným s umístěním hlavního spustitelného souboru `atpgui.exe`. Je třeba si ovšem dávat pozor na kolize jmen souborů z balíčku a souborů jiných balíčků, či přímo ATP GUI - například různé verze DLL knihoven.

4.8 Dynamické moduly

Modul má podobu dynamicky načítané sdílené knihovny. Seznam knihoven k použití ATP GUI je specifikován v textovém souboru `plugins.list`, každému modulu přísluší jedna řádka. Moduly jsou potom seřazeny ve stejném pořadí jako v tomto souboru v menu *Tools*.

4.8.1 Povinné funkce

Každý modul musí definovat alespoň funkce uvedené v následujícím seznamu. Další funkce jsou již plně na vůli tvůrce plug-inu.

poznámka Následující definice jsou v jazyce C, možnosti použití jiných jazyků jsem blíže nezkoumal.

- **gchar *get_name(void)** Vrací pointer na řetězec s názvem pluginu, který bude zobrazen v menu *Tools*. Jedná se o standardní céčkový řetězec ukončený nulovým znakem. Není vhodné alokovat pro tento řetězec paměť na heapu (např. pomocí funkce malloc), protože v ATP GUI potom bude odalokován až při ukončení programu operačním systémem.

- **is_result_editable(void)** Jednoduchá funkce vracející hodnotu `TRUE`, je-li výstupem pluginu soubor se syntaxí ATP GUI, tedy určen pro další zpracování. Funkce vrací `FALSE`, pokud je tomu jinak (například konverze do TPTP).

- **GtkTextBuffer *convert(GtkTextBuffer *buf)** Tato funkce se spustí po kliknutí na daný plug-in v menu, jako parametr dostane pointer na GtkTextBuffer aktuálního souboru (to je ten, jehož „tab" je otevřen) a jejím úkolem je vytvořit (naalokovat) nový `GtkTextBuffer` se zpracovaným obsahem a vrátit na něj ukazatel. Tento nový buffer bude ATP GUI otevřen v novém tabu. Pokud by tato funkce měla otevírat nějaká okna, je třeba, aby byla modální, jinak může mezi jeho otevřením a zavřením někdo zrušit vstupní `GtkTextBuffer` a může dojít k nepěknému pádu programu.

4.8.2 Poznámky ke kompilaci

Jelikož jsou moduly dynamicky načítané knihovny, musí být zkompilovány jako *Placement Independent Code*, tedy jako takový kód, u kterého nezáleží na umístění v paměti. Přinutit C kompilátor z GCC, aby takový kód produkoval lze pomocí parametru `-fPIC`. Někdy může být nutné přidat též parametr `-Wl,-export-dynamic`.

Modul je možné linkovat i proti dalším sdíleným knihovnám, v případě potřeby budou další knihovny načteny, pokud jsou k dispozici. Je úkolem tvůrce zajistit, aby případné další knihovny k dispozici byly, nejspíše kontrolou jejich přítomnosti při instalaci - na Linuxu nejspíše ve skriptu `configure`. Tvůrce by též měl zajistit alespoň krátky návod k instalaci v podobě souboru `<jméno pluginu>_README.txt`.

Mějme tedy zdrojový kód pluginu v souborech `plugin_file1.c` a `plugin_file1.c` a chceme vytvořit modul `libatpgui_example_plugin.so`. Příkazy pro kompilaci (a linkování) potom budou vypadat takto:

```
gcc ‘pkg-config --cflags gtk+-2.0‘ -fPIC \
  -o plugin_file1.o -g0 -c plugin_file1.c
gcc ‘pkg-config --cflags gtk+-2.0‘ -fPIC \
  -o plugin_file2.o -g0 -c plugin_file2.c
gcc -shared -Wl,-soname,libatpgui_example_plugin.so \
  -o libatpgui_example_plugin.so -Wl,-export-dynamic \
  plugin_file1.o plugin_file2.o ‘pkg-config --libs gtk+-2.0‘
```

Literatura

[1] William McCune, **Prover9 Reference Manual**
http://www.cs.unm.edu/ mccune/mace4/manual/April-2007/

[2] William McCune, **Otter v3.3f**
http://www-unix.mcs.anl.gov/AR/otter/

[3] William McCune, **Mace4**
http://www.cs.unm.edu/ mccune/mace4/manual/April-2007/

[4] Thomas Hillenbrand, Bernd Löchner, Arnim Buch, Andreas Jaeger, Roland Vogt, **Waldmeister**
http://waldmeister.org/

[5] Koen Claessen and Niklas Sörensson, **Paradox**
http://www.cs.chalmers.se/ koen/paradox/

[6] Stephan Schultz, **E**
http://www.eprover.org/

[7] GTK API Reference Manual
http://www.gtk.org/api/

[8] libXML2 Reference Manual
http://xmlsoft.org/html/index.html

[9] Free Software Foundation(1989), **GNU GPL version 2**, Copyright (C) 1989, 1991
http://www.gnu.org/licenses/gpl.html

[10] **Wikipedia - The Free Encyclopedia**
http://wikipedia.org/

[11] Sutcliffe, G. and Suttner, C.B., **CASC**
http://www.cs.miami.edu/ tptp/CASC/

[12] Koen Claessen and Niklas Sörensson, **New Techniques that Improve MACE-style Finite Model Finding**, Chalmers University of Technology and Göteborg University, Göteborg 2003

[13] Sutcliffe, G. and Suttner, C.B., **The TPTP Problem Library: CNF Release v1.2.1**, Miami 1998
http://tptp.org

[14] Štěpánek, P., **Predikátová logika**, Matematicko-fyzikální fakulta UK, Praha 2000

[15] Knuth, D.E. and Bendix, P.B. **Simple word problems in a universal algebra**
Computational Problems in Abstract Algebra (Ed. J. Leech) pages 263–297, 1970

[16] Bachmair, L.; Dershowitz, N. and Plaisted, D.A. **Completion without failure**
Resolutions of Equations in Algebraic Structures, volume 2, pages 1-30. Academic Press, 1989

[17] Furbach, U. and Obermaier, C. **Applications of Automated Reasoning**
Universität Koblenz-Landau, Koblenz