VIII ERMAC-R3 – 8º Encontro Regional de Matemática Aplicada e Computacional

20 a 22-Novembro-2008
Universidade Federal do Rio Grande do Norte - Natal/RN

Implementação de um gerador e verificador de modelos finitos para a Lógica Clássica de Primeira Ordem

Giuliano Vilela, Max Rosan, João Marcos (Orientador),
Departamento de Informática e Matemática Aplicada, CCET — UFRN, 59072-970, Natal – RN.
E-mails: giulianoxt@gmail.com, m.r650200@gmail.com, jmarcos@dimap.ufrn.br

Introdução

Neste trabalho, mostra-se o processo de desenvolvimento e o funcionamento interno de um gerador e verificador de modelos finitos para a Lógica Clássica de Primeira Ordem. Trata-se de uma aplicação web, implementando dois módulos do projeto **Logicamente**: um gerador exaustivo de modelos e um verificador simples, com o qual o usuário pode montar o seu próprio modelo. O objetivo deste trabalho foi investigar uma solução algorítmica para os problemas de *Model Checking* que abordamos. Discutiremos as diversas decisões de design da implementação que foram tomadas, e uma visão geral dos algoritmos utilizados.

Palavras-chave

Lógica Computacional, Bounded Model Checking.

Logicamente

O projeto **Logicamente** é uma suíte online de programas integrando vários módulos de software, em forma de aplicativos web, desenvolvidos por diversos alunos de graduação ao longo dos últimos dois anos, com trabalho voluntário de alunos da disciplina de "Lógica Aplicada à Computação" e financiamento parcial por parte da Pró-Reitoria de Graduação como um Projeto de Apoio à Melhoria da Qualidade do Ensino. Cada módulo implementa uma pequena funcionalidade da suíte, complementando o projeto.

O objetivo deste projeto é levar o conteúdo tradicional das disciplinas de graduação referentes à Lógica Computacional para a web, facilitando o processo de aprendizado dos alunos. Os aplicativos fazem uso de diversos recursos de webdesign utilizando a linguagem PHP e a plataforma Javascript.

Este trabalho se insere diretamente no contexto do **Logicamente**. Iremos apresentar o design e a implementação de dois módulos: *Model Builder* e *Model Checker*. Primeiramente, apresenta-se uma pequena descrição dos conceitos da Lógica Clássica de Primeira Ordem que serão utilizados.

Lógica Clássica de Primeira Ordem

Também conhecida como *Cálculo de Predicados*, a *Lógica Clássica de Primeira Ordem* – LCPO, é um sistema lógico formal que estende a *Lógica Clássica Proposicional* – LCP, adicionando a noção de átomos estruturados (contendo *predicados* e *funções*) e de *quantificadores* sobre variáveis, dando vezo assim a uma maior riqueza semântica.

Por exemplo: a afirmação "*chove em certo dia de um certo mês*" dificilmente poderia ser traduzida para a LCP, devido à escassez, nesta última, de recursos sintáticos e semânticos apropriados. Ao contrário, na LCPO, há uma tradução simples, utilizando as noções de predicados e de quantificadores:

$$\exists d\, \exists m\, (Pertence(d, m) \wedge Chove(d))$$

Devido a estes recursos, a LCPO é uma linguagem com várias extensões e diversas aplicações reais. Com esta Lógica, é possível formalizar a teoria dos conjuntos de Zermelo-Fraenkel, como também a aritmética de Peano, teorias que podem ser tomadas como os pilares da matemática clássica.

Sintaxe da LCPO

Para cada contexto em que se utilize a LCPO, é possível fixar um conjunto $\Sigma = \langle SF, SP\rangle$ de símbolos sintáticos, conhecido por *assinatura*, onde SF e SP são conjuntos de símbolos que representam, respectivamente, funções e predicados (com aridades conhecidas). Considere também X, um conjunto infinito, enumerável, de símbolos de variáveis.

É possível definir um novo conjunto de símbolos, denominado *alfabeto*, sobre Σ e X, representado por Alf_Σ^X. Têm-se que: $Alf_\Sigma^X = (SF \cup SP \cup X \cup \{\top, \bot, \leftrightarrow, \rightarrow, \wedge, \vee, \forall, \exists, `,', `(', `)'\})$. O alfabeto de primeira ordem define uma lista de símbolos sintáticos que podem aparecer em uma fórmula da LCPO.

Para representar termos arbitrários da LCPO define-se o conjunto T_Σ^X sobre Alf_Σ^X, denominado conjunto de *termos induzidos por* Alf, da seguinte forma:

- $cst \in T_\Sigma^X$, onde $cst \in SF$ é um símbolo de função 0-ária. Consideramos as funções com esta aridade como sendo *constantes*.
- $x \in T_\Sigma^X$, onde $x \in X$.
- $\{t_1, \ldots, t_n\} \subseteq T_\Sigma^X \Rightarrow f(t_1, \ldots, t_n) \in T_\Sigma^X$, onde $f \in SF$ é um símbolo de função n-ária.

Podemos, então, definir o conjunto $Ling_{LCPO}{}^{X}_{\Sigma}$, ou simplesmente $Ling_{LCPO}$, que contém todas as fórmulas bem formadas da LCPO, fixada uma assinatura $\Sigma = \langle SF, SP\rangle$ e um conjunto X de variáveis. Sejam $\{t_1, \ldots, t_n\} \subseteq T^X_\Sigma$, $P^0 \in SP$ um predicado 0-ário, $P^n \in SP$ (com $n \neq 0$) um predicado n-ário e $\varphi, \varphi_0, \varphi_1 \in Ling_{LCPO}$, têm-se:

- $P^0 \in Ling_{LCPO}$
- $P^n\,(t_1, \ldots, t_n) \in Ling_{LCPO}$
- $\{\forall x\ \varphi, \exists x\ \varphi\} \subseteq Ling_{LCPO}$
- $\{\bot, \top, \neg\varphi\} \subseteq Ling_{LCPO}$
- $\{(\varphi_0 \to \varphi_1), (\varphi_0 \wedge \varphi_1), (\varphi_0 \vee \varphi_1), (\varphi_0 \leftrightarrow \varphi_1)\} \subseteq Ling_{LCPO}$

É interessante observar que as fórmulas da LCP são um subconjunto das fórmulas da LCPO. Basta considerar os átomos da LCP como sendo predicados 0-ários de primeira ordem. A definição de $Ling_{LCPO}$ dada acima tem um forte caráter recursivo, e pode ser diretamente mapeada para uma definição computacional em forma de árvore.

Semântica da LCPO

Para definir a semântica da LCPO, introduziremos diversos conceitos separadamente.

Modelo

Dada uma assinatura $\Sigma = \langle SF, SP\rangle$ da LCPO, é possível fixar uma *estrutura de interpretação*, também denominada de *modelo*, $M = \langle U, I\rangle$, onde U é um conjunto não vazio de objetos, denominado *universo de discurso* e I é uma *função de interpretação*.

Cada símbolo de SF e SP está associado, através de I, a uma estrutura matemática real, que é dita a sua *interpretação*. Assim:

- $r \in SP \Rightarrow I(r) \in \mathrm{Pow}(U^n)$, onde r é um símbolo de predicado n-ário. $I(r)$ é uma *relação*.
- $c \in SF \Rightarrow I(c) \in U$, onde c é um símbolo de função 0-ário.
- $f \in SF \Rightarrow I(f) \in \mathrm{Pow}(U^{n+1})$, onde f é um símbolo de função n-ário (com $n \neq 0$). $I(f)$ é uma *função n-ária*, um caso especial de relação $(n+1)$-ária onde existe uma relação unívoca entre os n primeiros argumentos e o último argumento de f.

Um aspecto interessante explorado pelo *Model Checker* é que, para uma mesma fórmula φ, que só depende de uma assinatura Σ para ser bem formada, é possível interpretá-la de diferentes maneiras, dependendo do modelo considerado.

Atribuição

Uma atribuição $\rho : X \longrightarrow U$ baseada num modelo $M = \langle U, I\rangle$ é uma função que associa símbolos de variáveis a objetos do universo de discurso. Uma atribuição ρ' é dita x-equivalente a outra atribuição ρ caso $\rho'(y) = \rho(y)$ para toda variável y tal que $y \neq x$.

Utilizando as noções de modelo e de atribuição podemos estender a idéia de interpretação de predicados e funções para termos em geral. A interpretação de $t \in T^X_\Sigma$ em uma atribuição ρ e modelo $M = \langle U, I\rangle$, representada por $[t]^\rho_M$ é dada pelas seguintes propriedades:

- $[x]^\rho_M = \rho(x)$, onde $x \in X$.
- $[c]^\rho_M = I(c)$, onde $c \in SF$ é uma função 0-ária.
- $[f(t_1, \ldots, t_n)]^\rho_M = I(f)([t_1]^\rho_M, \ldots, [t_n]^\rho_M)$, onde $f \in SF$ é uma função n-ária e $\{t_1, \ldots, t_n\} \subseteq T^X_\Sigma$.

Satisfatibilidade de fórmulas

Dado um modelo $M = \langle U, I\rangle$ sobre uma assinatura $\Sigma = \langle SF, SP\rangle$ e uma atribuição $\rho : X \longrightarrow U$, iremos definir a noção de satisfação de uma fórmula φ por um modelo e atribuição, denotada por $M, \rho \Vdash \varphi$.

Sejam $P^0, P^n \in SP$ símbolos de predicados 0-ário e n-ário, respectivamente (com $n \neq 0$) e $\varphi_n \in Ling_{LCPO}$, têm-se:

- $M, \rho \Vdash P^0$, caso $I(P^0)$ seja o caso.
- $M, \rho \Vdash P^n(t_1, \ldots, t_n)$, caso $I(P^n)([t_1]^\rho_M, \ldots, [t_n]^\rho_M)$ seja o caso, onde $\{t_1, \ldots, t_n\} \subseteq T^X_\Sigma$.
- $M, \rho \nVdash \bot$ e $M, \rho \Vdash \top$
- $M, \rho \Vdash \neg\varphi$, caso $M, \rho \nVdash \varphi$.
- $M, \rho \Vdash (\varphi_0 \to \varphi_1)$, caso $M, \rho \nVdash \varphi_0$ ou $M, \rho \Vdash \varphi_1$.
- $M, \rho \Vdash (\varphi_0 \leftrightarrow \varphi_1)$, caso $M, \rho \Vdash \varphi_0 \to \varphi_1$ e $M, \rho \Vdash \varphi_1 \to \varphi_0$.
- $M, \rho \Vdash (\varphi_0 \wedge \varphi_1)$, caso $M, \rho \Vdash \varphi_0$ e $M, \rho \Vdash \varphi_1$.
- $M, \rho \Vdash (\varphi_0 \vee \varphi_1)$, caso $M, \rho \Vdash \varphi_0$ ou $M, \rho \Vdash \varphi_1$.
- $M, \rho \Vdash \forall x\ \varphi$, caso para todo ρ' x-equivalente a ρ, tem-se $M, \rho' \Vdash \varphi$.
- $M, \rho \Vdash \exists x\ \varphi$, caso exista ρ' x-equivalente a ρ tal que $M, \rho' \Vdash \varphi$.

Com estes conceitos, podemos abordar a arquitetura dos módulos implementados e nossas decisões de design.

Implementação do gerador e verificador de modelos

Iremos descrever o design e a implementação do cerne dos módulos *Model Checker* e *Model Builder* do **Logicamente**. Primeiramente, veremos as estruturas de dados utilizadas para representar alguns conceitos da LCPO. Depois, veremos o algoritmo de decisão para a satisfatibilidade de fórmulas sob um modelo, e o algoritmo de geração exaustiva de modelos para uma dada assinatura.

Representação das fórmulas

Fazendo parte do núcleo do **Logicamente** está o módulo *Formula Reader*. Este módulo é capaz de ler fórmulas em forma de string e realizar o parsing para uma representação em forma de árvore.

O parser assume a assinatura $\Sigma = \langle SF, SP\rangle$ e conjunto de variáveis X para as fórmulas de entrada, onde: $SF = \{a, b, c, d, e, f, g, h\}$, $SP = \{P, Q, R\}$ e $X = \{x, y, z, w\}$. A gramática que descreve as fórmulas válidas do *Formula Reader* é a seguinte:

```
<Variável>      ::= x | y | z | u | v

<Constante>     ::= a | b | c | d | e

<SimboloFunc>   ::= f | g | h

<SimboloPred>   ::= P | Q | R

<Quantificador> ::= A | E

<ConectivoUn>   ::= ~

<ConectivoBin>  ::= --> | <-> | & | '|'

<Termo>     ::=
    <Variável> |
    <Constante> |
    <SimboloFunc>(<Termo>,...,<Termo>)

<Predicado> ::=
    <SimboloPred> |
    <SimboloPred>(<Termo>,...,<Termo>)

<Formula>   ::=
    <Predicado> |
    <ConectivoUn> <Formula> |
    (<Formula> <ConectivoBin> <Formula>) |
    <Quantificador> <Variável> <Formula>
```

Seguem alguns exemplos de fórmulas da LCPO escritas com caracteres ASCII de acordo com a gramática acima. Seja $\varphi \in Ling_{LCPO}$ e φ' sua representação no *Formula Reader*, tem-se:

- Se $\varphi = \forall x\ P(x)$, então $\varphi' = \mathrm{A}x\ P(x)$.
- Se $\varphi = (P \vee \exists x\ Q(x)) \rightarrow \exists x(P \wedge Q(x))$ então $\varphi' = ((P \mid \mathrm{E}x\ Q(x)) --> \mathrm{E}x\ (P\ \&\ Q(x))$.

O resultado do processo de parsing do *Formula Reader* é uma árvore, cuja estrutura pode ser definida indutivamente, de forma semelhante ao que foi apresentado na seção de sintaxe da LCPO. A figura [1] mostra, resumidamente, as classes que compõem as árvores.

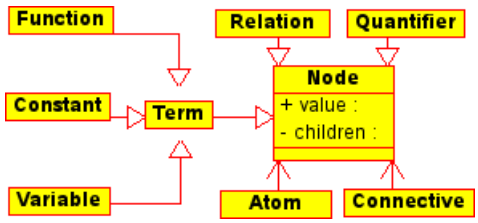


Figura 1: Diagrama UML de classes do *Formula Reader*

Tanto o *Model Checker* quanto o *Model Builder* utilizam esta estrutura para fazer o parsing das fórmulas internamente. Por exemplo, para a fórmula $\mathrm{A}xR(x)$, uma versão simplificada da árvore gerada seria:

```
Quantifier (
  value => ('A', Variable(value => 'x'))
  children =>
    Relation(value => 'R',
             children =>
               Variable(value => 'x')
    )
)
```

Esta representação se mostrou bastante eficiente para os algoritmos que empregamos, devido à maneira natural com que suporta o percurso recursivo ao longo da fórmula.

Representação das assinaturas

Dada uma fórmula φ, uma das tarefas necessárias para o algoritmo de geração de modelos é extrair a assinatura mínima correspondente a φ, isto é, o menor alfabeto que contenha todos os símbolos que ocorrem em φ.

Esta é uma tarefa simples, realizada através de um percurso ao longo da árvore que representa φ, armazenando informações sobre as relações, funções e constantes utilizadas. As informações são armazenadas em um array, de forma sequencial e sem qualquer ordem particular.

Representação das atribuições

Dada um conjunto X, enumerável, de variáveis, e um universo de discurso U, representamos uma atribuição $\rho : X \longrightarrow U$ como uma tabela *hash* de strings para inteiros. Assim, o símbolo que representa as variáveis é diretamente mapeado para um inteiro, que seria um elemento de U.

Representação dos modelos

Como vimos na seção sobre a semântica da LCPO, um modelo $M = \langle U, I\rangle$ é definido levando em conta uma assinatura Σ. A representação que escolhemos

não leva em conta as características de uma fórmula específica, e sim dos elementos de assinatura. Dessa maneira, Σ determina uma classe de modelos estruturalmente compatíveis com todas as fórmulas φ construídas sobre essa assinatura.

Por convenção, o universo de discurso U será sempre da forma $\mathbb{N}_S = \{n \in \mathbb{N} : n < S\}$, onde $S = |U|$ é o tamanho de universo a ser considerado, informado previamente pelo usuário.

O modelo é representado como uma tabela *hash* de strings para outras estruturas. As constantes são mapeadas para inteiros simples e as funçoes e relações são mapeadas para arrays multi-dimensionais.

Considere uma relação R de aridade 3, onde $R = \{\langle 0,1,0\rangle, \langle 1,0,1\rangle, \langle 1,1,1\rangle\}$. Internamente, modelamos R como uma árvore S-ária (através de um array multi-dimensional), onde S é o tamanho do universo de discurso. Esta árvore representa a função característica de R, isto é:

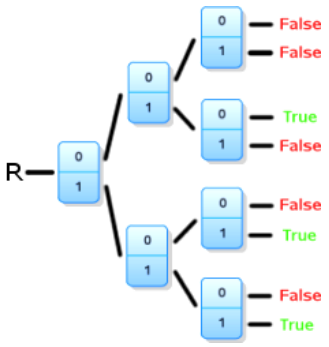


Figura 2: Representação interna da relação R

R seria, então, uma referência para o nó raiz da árvore. Equivalentemente, podemos ver R como uma referência para um array contendo outras referências, e assim sucessivamente, até chegar-se no último array, que contém valores booleanos. Esses valores indicam a pertinência de triplas ao conjunto R. Assim, utilizando uma notação adequada para arrays multi-dimensionais (neste caso, tridimensionais), $R[0][1][0] = R[1][0][1] = R[1][1][1] =$ `true`.

Para funções, a representação é bastante semelhante. A única mudança é nos valores guardados nas folhas da árvore, onde em vez de armazenar valores booleanos, armazenam-se inteiros. Assim, dada uma função f de aridade 3, $f[1][0][1]$ seria o inteiro x tal que $\langle 1,0,1,x\rangle \in f$.

A eficiência desta representação é questionável. Poderíamos simplesmente guardar uma lista com as ênuplas pertencentes à relação (ou função) em questão. Esta seria uma opção válida caso a relação fosse constante ao longo do tempo, porém a representação que escolhemos é interessante para o algoritmo de geração de modelos, onde teremos que variar o conteúdo da relação.

Apresentam-se agora os algoritmos que foram desenvolvidos para a tarefa de geração dos modelos e de verificação de satisfabilidade. Eles trabalham com modelos, naturalmente, finitos. Assume-se o universo de discurso implícito $\mathbb{N}_S$, tal como descrito anteriormente.

Algoritmo de geração exaustiva de modelos

Dada uma fórmula φ, extrai-se dela o conjunto Σ, sua assinatura mínima correspondente. Como já foi observado, os possíveis modelos que satisfazem esta fórmula não dependem estruturalmente de φ, somente de sua assinatura. Desse modo, os algoritmos de geração trabalham sob a assinatura Σ.

Versão recursiva

A geração se dá com um percurso recursivo ao longo do vetor que representa a assinatura Σ. Assuma um conjunto `Models`, inicialmente vazio. Dado um natural $S \in \mathbb{N}$, ao final do algoritmo `Models` irá conter todos os modelos $M = \langle U, I\rangle$ para a assinatura Σ, onde $|U| = S$. A chamada inicial a esta rotina é `generateModels(`Σ`, S, 1, {})`.

```
def generateModels(Σ, S, pos, M)
if pos = |Σ| then
  Models := Models ∪ {M}
else
  element := Σ[pos]
  if element is Constant then
    for i ∈ (0 .. (S − 1)) do
      M[element.symbol] := i
      generateModels(Σ, S, pos+1, M)
    end for
  else if element is Relation then
    relations := genRelations(element.arity, S)
    for r ∈ relations do
      M[element.symbol] := r
      generateModels(Σ, S, pos+1, M)
    end for
  else if element is Function then
    functions := genFunctions(element.arity, S)
    for f ∈ functions do
      M[element.symbol] := f
      generateModels(Σ, S, pos+1, M)
    end for
  end if
end if
```

Por simplicidade, omitimos as definições das funções auxiliares `genRelations` e `genFunctions`. São funções recursivas que retornam um conjunto com todas as possíveis relações e funções, respectivamente, para uma dada aridade.

Versão iterativa

A geração iterativa de modelos, mais eficiente, não armazena todos os possíveis modelos de uma vez. A idéia é que os modelos são gerados um a um, de maneira sequencialmente ordenada, aumentando a responsividade do sistema.

A idéia deste algoritmo é transformar uma estrutura que representa, por exemplo, uma relação em um número em uma certa base numérica. Seja $S \in \mathbb{N}$ o tamanho do universo informado pelo usuário, $r \in \mathbb{N}$ a aridade de uma certa relação, e $\mathbb{N}_L = \{x \in \mathbb{N} : x < 2^{(S^r)}\}$. Montamos uma função bijetora $Flat : \text{Pow}(\mathbb{N}_S \times \cdots \times \mathbb{N}_S) \longrightarrow \mathbb{N}_L$, e também a sua inversa $Flat^{-1}$. $Flat$ representa um algoritmo que utilizamos para converter uma representação de arrays multi-dimensionais para um número inteiro.

Considerando a relação R, cuja representação em arrays foi mostrada na figura [2]. Tem-se que $Flat(R)$ é o número cuja representação binária é '00100101'. Note que os zeros representam os valores *False* da figura, enquanto os uns representam os valores *True*.

Logo, a essência do algoritmo iterativo é que, começando do número 0 e fazendo somas sucessivas adicionando uma unidade à representação $Flat$ de uma certa relação R, iremos encontrar todas as possíveis relações com a mesma aridade sobre o mesmo universo de discurso de R. Aplicando este procedimento também para constantes e para funções, com algumas adaptações (trabalhando com números não binários, mas na base S), garantimos que a geração de modelos é exaustiva para um certo tamanho finito do universo de discurso, definido pelo usuário.

Seguem as rotinas que implementam este algoritmo: `addOne` é responsável por adicionar 1 ao número representado pelo vetor `vet` (vetor de dígitos), retornando `true` caso a soma tenha sido um sucesso ou `false` caso `vet` seja o maior número possível.

```
def addOne(vet, base)
for i ∈ (0 .. (|vet| − 1)) do
  if vet[i] < base then
    vet[i] := vet[i] +1
    return true
  else
    vet[i] := 0
  end if
end for
return false
```

A essência do outro algoritmo, `nextModel`, é que, fixado o tamanho S do universo de discurso, dado um modelo $M_n = \langle U, I_n \rangle$, pode-se gerar o próximo modelo $M_{n+1} = \langle U, I_{n+1} \rangle$ a partir de M_n. Iniciando esta cadeia, o modelo M_0 é gerado de forma especial: todas as contantes são mapeadas para o número 0, todas as relações são mapeadas para conjuntos vazios e todas as funções são funções totais onde o conjunto imagem é $\{0\}$. A partir de M_0, as chamadas de `nextModel` retornam `true` caso exista um próximo modelo (e este será armazenado em M) ou retornam `false` caso o modelo M seja o último para este tamanho de universo.

```
def nextModel(M, Σ, S)
const := conjunto imagem dos mapeamentos de constantes em M
b := addOne(const, S − 1)
M := M com os novos mapeamentos de constantes em const
if b = true then
  return true
end if
for r ∈ Σ, onde r é uma relação do
  rel := Flat(M[r])
  b := addOne(rel, S − 1)
  M[r] := Flat^{-1}(rel)
  if b = true then
    return true
  end if
end for
for f ∈ Σ, onde f é uma função do
  func := Flat(M[f])
  b := addOne(func, S − 1)
  M[f] := Flat^{-1}(func)
  if b = true then
    return true
  end if
end for
return false
```

Algoritmo de decisão para a satisfatibilidade de fórmulas da LCPO

Dado um modelo $M = \langle U, I \rangle$, uma atribuição ρ e uma fórmula φ, este algoritmo é responsável por fazer uma análise *top-down* da árvore que representa φ, a fim de determinar se $M, \rho \Vdash \varphi$ é o caso.

A chamada `checkModel`(φ, M, $\emptyset$, S) retorna `true` caso $M, \emptyset \Vdash \varphi$. Note que a cadeia de **if**'s utilizada no algoritmo apresenta uma semântica de casamento de padrões entre duas árvores que representam a mesma fórmula. Por exemplo, a expressão $\varphi = \varphi_0 \wedge \varphi_1$ indica que φ é, estruturalmente, uma conjunção entre duas outras fórmulas. Este algoritmo é fortemente baseado nas propriedades de $\Vdash$ para a LCPO que foram enunciadas anteriormente.

Com isto, acabamos de cobrir toda a arquitetura de software utilizada pelo *Model Checker* e pelo *Model Builder*. Iremos agora analisar os recursos dos módulos em mais detalhes, assim como as interfaces web.

Funcionalidade dos módulos

Os módulos apresentados neste trabalho utilizam a base de software que foi apresentada na seção anterior. Primeiramente, abordaremos o *Model Builder*, e logo após o *Model Checker*.

O procedimento executado pelo *Model Builder* é o seguinte: dada uma expressão φ, o primeiro passo a ser feito é convertê-la para uma árvore sintática, ou seja, é realizada a análise sintática (parsing) da expressão, utilizando o *Formula Reader*. Após a geração da árvore, é realizada uma *busca em ordem* para identificar a assinatura mínima correspon-

```
def checkModel(φ, M, ρ, S)
if φ = ∀x φ' then
  for i ∈ (0 .. (S − 1)) do
    ρ[x] := i
    if checkModel(φ', M, ρ, S) = false then
      return false
    end if
  end for
  return true
else if φ = ∃x φ' then
  for i ∈ (0 .. (S − 1)) do
    ρ[x] := i
    if checkModel(φ',M,ρ,S) = true then
      return true
    end if
  end for
  return false
else if φ = (φ0 → φ1) then
  return
  (not checkModel(φ0,M,ρ,S)) or
  checkModel(φ1,M,ρ,S)
else if φ = (φ0 ↔ φ1) then
  return
  checkModel(φ0,M,ρ,S) =
  checkModel(φ1,M,ρ,S)
else if φ = (φ0 ∧ φ1) then
  return
  checkModel(φ0,M,ρ,S) and
  checkModel(φ1,M,ρ,S)
else if φ = (φ0 ∨ φ1) then
  return
  checkModel(φ0,M,ρ,S) or
  checkModel(φ1,M,ρ,S)
else if φ = ¬φ' then
  return not checkModel(φ',M,ρ,S)
else if φ = ⊥ then
  return false
else if φ = ⊤ then
  return true
else if φ = R(t1, ..., tn) then
  return M[R][[t1]^ρ_M] ... [[tn]^ρ_M] = true
end if
```

dente a φ. Uma vez identificada, a assinatura Σ é utilizada para a geração de todos os possíveis modelos, de acordo com o tamanho máximo do universo de discurso, que é informado pelo usuário. O algoritmo, então, verifica os modelos de acordo com a sua ordem sequencial de geração e pára quando encontrar um modelo M que satisfaça a fórmula ou quando o tamanho do universo de discurso do modelo atual for maior que o tamanho informado pelo usuário.

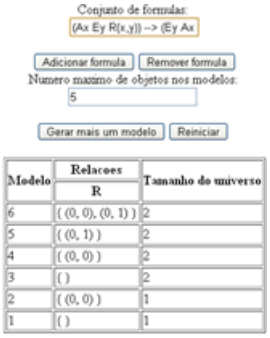
Seja L uma lista de modelos gerados pelo algoritmo e S o tamanho máximo dos universos de discurso, informado pelo usuário. L apresenta todas as possibilidades de modelos válidos que possuem o universo U com cardinalidade menor ou igual a S. Para qualquer elemento da assinatura, todas as possibilidades de interpretação são combinadas com todas as possibilidades de interpretação dos outros elementos da assinatura, gerando-se no final todos os possíveis modelos. Resumindo, L obedece a seguinte propriedade: para todo modelo válido $M = \langle U, I\rangle$, correspondente à assinatura Σ, se $|U| \leq A$ então $M \in L$.

O procedimento aplicado pelo *Model Builder* toda vez que o usuário requisita um novo modelo seria:

```
Seja M = ⟨U, I⟩ o primeiro elemento de L
while M, ∅ ⊮ φ do
  if |U| > Tam then
    print "Não há mais modelos nestas condições
    que satisfaçam φ."
    return
  else
    M := próximo elemento de L
  end if
end while
adiciona uma linha correspondente à M na tabela de
modelos
```

A versão online do *Model Builder* utiliza a implementação iterativa do algoritmo de geração exaustiva de modelos. Logo, a lista L nunca é armazenada em totalidade na memória do servidor web. Ao invés disso, seus elementos são criados um a um, de acordo com a demanda do usuário.

A interface principal do *Model Builder* é composta por um campo para o usuário digitar a expressão, dois botões e uma tabela para saída dos resultados.



A fórmula a ser verificada deve ser digitada no campo *Conjunto de formulas*, que guarda uma fila de fórmulas. Caso o usuário deseje verificar mais de uma fórmula deve clicar em *Adicionar formula* para adicionar um campo para uma nova fórmula, que seria adicionada ao fim da fila. Ao digitar um conjunto de fórmulas, a expressão a ser verificada será a conjunção das fórmulas digitadas. Já ao clicar no botão *Remover formula*, a primeira fórmula da fila é removida. No campo *Número máximo de objetos nos modelos*, o usuário deve digitar o tamanho máximo desejado do universo de discurso dos modelos gerados.

O modelo gerado aparece de forma iterativa e logo abaixo em uma tabela. Essa iteração é realizada através do botão *Gerar mais um modelo* que, ao ser pressionado, faz com que mais um modelo seja

buscado e exibido. O clique no botão *Reiniciar* resulta no recomeço da busca por modelos em que o universo de discurso volta a ter tamanho 1, como no início. A tabela é dividida em colunas, onde a coluna *Modelo* informa a ordem de geração do modelo, *Tamanho do universo* informa o tamanho do universo do modelo e as outras colunas informam a interpretação dada pelo modelo.

Com o *Model Checker*, apresentado acima, o usuário pode criar um modelo M e testar se ele satisfaz uma certa fórmula φ. Sua interface possui apenas dois campos: um para o usuário digitar a fórmula e outro para informar o tamanho do universo do seu modelo. De forma semelhante ao *Model Builder*, no *Model Checker* subentende-se que o universo de discurso é um subconjunto dos naturais e que o maior natural pertencente a esse conjunto é o antecessor direto daquele que representa a cardinalidade do universo de discurso. Ao clicar no botão *Gerar assinatura*, serão gerados campos logo abaixo correspondendo aos elementos da assinatura da fórmula. Em cada campo gerado, o usuário deve informar o valor que o elemento correspondente deverá assumir, no caso de variáveis livres e constantes, ou quais ênuplas devem pertencer ao conjunto, no caso de relações ou funções. Após construir seu modelo, o usuário pode clicar em *Checar modelo* para verificar se o modelo satisfaz a formula digitada no campo *Fórmula*.

Conclusão

O problema da decidibilidade da Lógica Clássica de Primeira Ordem para modelos com universos de discurso possivelmente infinitos, conhecido pelo nome de *"Entscheidungsproblem"* (tradução alemã para *"problema de decisão"*), foi um desafio proposto por David Hilbert em 1928. Nos anos de 1936 e 1937 foram publicados artigos independentes por Alonzo Church e Alan Turing contendo demonstrações equivalentes da não existência de uma solução geral para este problema. Neste trabalho, entretanto, investigamos uma solução algorítmica para uma versão restrita deste problema: a decidibilidade da LCPO utilizando modelos finitos com cardinalidade pré-definida.

Mostramos como se deu a implementação de um gerador e verificador de modelos finitos com cardinalidade pré-definida para a LCPO. Foram abordados conceitos dessa lógica necessários ao entendimento do assunto e também as decisões de design e de implementação que foram tomadas ao longo do desenvolvimento. Com este pequeno projeto, conseguimos estender os módulos do **Logicamente** para trabalhar com a Lógica de Primeira Ordem, investigando algoritmos para o problema de verificação de modelos limitados (bounded model checking) para a LCPO.

Referências

[1] **Logicamente**, aplicativo web disponível em `http://www.dimap.ufrn.br/logicamente`

[2] B. R. Bedegral, B. M. Acióly, "Introdução à Lógica Clássica para a Ciência da Computação". Versão preliminar, Natal, Junho 2007.

[3] P. Gouveia, F.M. Dionísio, J. Marcos, "Lógica Computacional". Versão preliminar, 2007.

[4] J. Jeuring, D. Swierstra. "Grammars and Parsing", 2001.

[5] "Análise Sintática (Computação)". Disponível em *http://pt.wikipedia.org/wiki/Análise_sintática*. Acessado em: 21 de outubro de 2008.