Protótipo de interpretador para cTVQL

Rodrigo Machado , Álvaro Freitas Moreira

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{rma,afmoreira}@inf.ufrgs.br

Resumo. *Este trabalho apresenta um interpretador e avaliador de tipos construído a partir da descrição formal da linguagem de consulta de um modelo de banco de dados temporal e versionado. A construção de protótipos de interpretadores a partir de especificações formais permite que haja uma experiência prática de programação sobre novos modelo de dados e linguagens de consulta, sendo interessante como framework a ser utilizado na fase de projeto de extensões a modelos e linguagens tradicionais.*

1. Introdução

Na área de banco de dados existem várias propostas que tratam de extensões dos modelos tradicionais a fim de suprir suporte a características requisitadas por certas aplicações. Entre tais características pode-se citar o suporte a estados alternativos para entidades (versões) e o registro das alterações dos dados ao longo do tempo. O Modelo Temporal de Versões (TVM) [Moro 2001] implementa tais características sob uma base orientada a objetos. A linguagem de consulta TVQL [Moro et al. 2002], por sua vez, permite que se realizem consultas sobre o modelo TVM.

A linguagem TVQL é interessante porque permite especificar restrições temporais versionados para a consulta e acessar estados alternativos de objetos. Ela representa um foco de interesse em pesquisa de bancos de dados temporais, para os quais ainda não há um padrão estabelecido. Até o presente momento, TVQL não possui interpretador.

Em [Machado 2005] é proposto um modelo formal para estudar a linguagem TVQL. A linguagem TVQL foi restrita a um núcleo denominado cTVQL, sobre o qual foi definida semântica operacional e se especificou o sistema de tipos. Através de tais especificações pode-se investigar formalmente propriedades estruturais da linguagem e de possíveis extensões.

Apesar da motivação da formalização de TVQL ter sido o estudo de propriedades, a utilização de sistemas de dedução como método de especificação pode proporcionar uma abstração confortável para a implementação de interpretadores experimentais. Por exemplo, se forem utilizadas linguagens que fazem uso de casamento de padrões, como Ocaml ou Haskell, pode-se obter facilmente um algoritmo de avaliação de consultas diretamente a partir da semântica operacional.

Neste trabalho, será apresentada a implementação de um interpretador e avaliador de tipos para a linguagem cTVQL, desenvolvido simultaneamente à formalização descrita em [Machado 2005]. Na Sessão 2 serão apresentados através de exemplos algumas características do modelo TVM e das linguagens TVQL e cTVQL. Na Sessão 3 será discutida a arquitetura do interpretador e suas características. Na Sessão 4 são apresentadas as considerações finais.

2. TVM e TVQL

A idéia fundamental por trás do modelo TVM é a possibilidade de haver objetos com *estados alternativos*, modificados ao longo do tempo. Para ilustrar, considere a seguir a seguinte declaração de classe TVM:

```
class Pessoa hasVersions
{
  nome : string
  temporal endereco : string
  temporal telefone : string
}
```

A cláusula `hasVersions` indica que a classe é *temporal versionada*. Isto significa que os objetos dessa classe podem manter estados alternativos (versões) e podem conter atributos temporais. Na classe `Pessoa`, são definidos três atributos, `nome`, `endereco` e `telefone`, sendo os dois últimos *temporais*. A diferença entre um atributo normal e um atributo temporal está principalmente na semântica da atualização de valores. Ao se atualizar o valor de uma atributo marcado como temporal, o valor anterior e o instante da troca de valores são guardados pelo SGBD. Desta forma, pode-se obter o estado de um mesmo objeto em diversos pontos do tempo. Ao se atualizar um atributo normal, o antigo valor é perdido definitivamente.

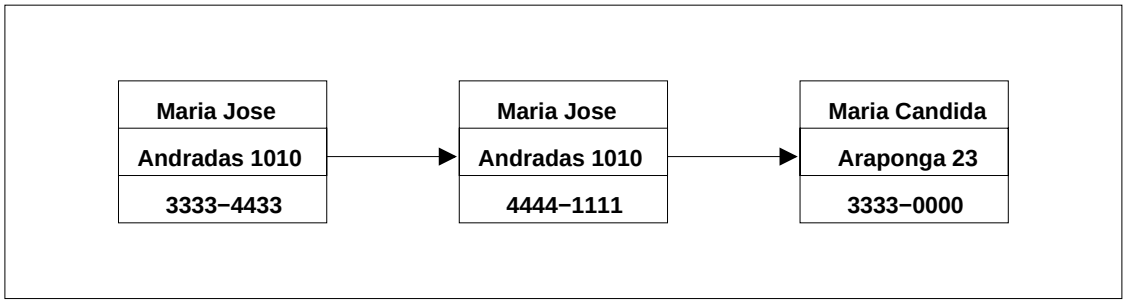

Figura 1. Modificações sobre um objeto temporal versionado.

Considere as modificação sobre um dado objeto da classe `Pessoa` apresentadas na Figura 1. A primeira modificação altera o valor de `telefone`, guardando o valor antigo. Na segunda modificação, todos os atributos são modificados e os valores antigos de `endereco` e `telefone` são guardados. O nome `'Maria Jose'` é perdido, já que o atributo `nome` é normal.

A linguagem TVQL, utilizando palavras chave como `ever` e `present`, possibilita resgatar o histórico das alterações dos objetos. Por exemplo, a consulta

```
select x.nome
from Pessoa x
where ever x.endereco = 'Andradas 1010'
```

retorna todos os nomes atuais de pessoas que um dia moraram no endereço `'Andradas 1010'`, no caso, {`'Maria Candida'`}. Por sua vez, a consulta

```
select ever x.telefone
from Pessoa x
where present x.nome = 'Maria Candida'
```

retorna todos os telefones um dia associados a pessoas cujo nome atual é `'Maria Candida'`, no caso, {`'3333-4433'`, `'4444-1111'`, `'3333-0000'`}.

A linguagem TVQL prevê que valores temporais e valores não-temporais sejam operados entre si sem distinções, o que pode se tornar um problema para o sistema de tipos. A linguagem cTVQL propõe algumas modificações estruturais na forma das consultas TVQL e na representação de valores temporais para contornar essa dificuldade, mantendo a mesma semântica das palavras chave `ever` e `present`. A última consulta é escrita em cTVQL da seguinte forma:

```
ever (select x.telefone
      from Pessoas x
      where present (x.nome = 'Maria Candida') )
```

As modificações propostas na estrutura de TVQL são descritas com mais detalhe em [Machado 2005]. A linguagem cTVQL foi definida usando como base OQL [Cattell et al. 2000], diferentemente de TVQL, baseada em SQL.

3. Interpretador cTVQL

3.1. Arquitetura

Por se tratar de uma linguagem de consulta, é necessário que o interpretador de consultas e o avaliador de tipos tenham acesso aos objetos armazenados no banco de dados. É importante também garantir que o estado dos objetos armazenados efetivamente siga as restrições impostas pelo esquema de classes, caso contrário, o avaliador de tipos não pode prever com segurança os tipos dos valores armazenados em atributos a partir do esquema. Assim sendo, o interpretador cTVQL possui três entradas de dados:

- *Query:* a consulta a ser avaliada;
- *State:* descrição do estado das extensões e objetos do banco de dados;
- *Schema:* definição das classes de objetos do SGBD.

Cada entrada é descrita textualmente utilizando-se um formato específico, sendo a representação interna construída a partir de *parsers*. O interpretador em si é composto por três módulos principais, apresentados a seguir:

- *Query Evaluator:* avalia a consulta até se chegar a um resultado final;
- *Type Checker:* determina um tipo específico para cada consulta, se possível;
- *Integrity Checker:* verifica a boa formação do estado de banco de dados em relação ao esquema.

A arquitetura do interpretador cTVQL é apresentada na Figura 2. A seguir, serão descritos com um pouco mais de detalhe os principais módulos.

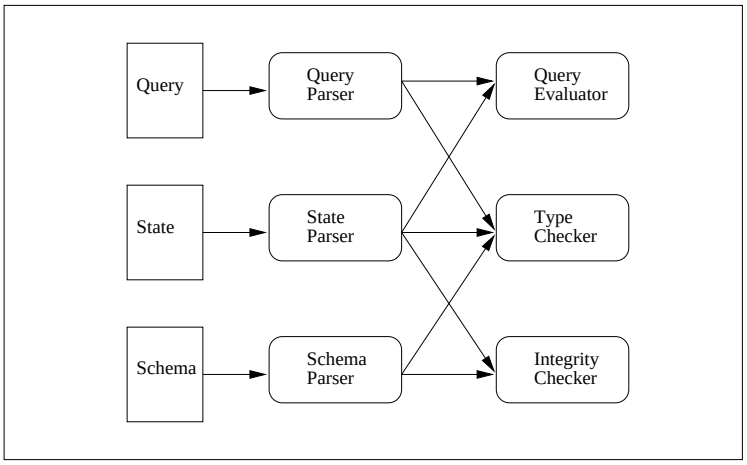

Figura 2. Arquitetura do Interpretador cTVQL.

3.2. Avaliador de consultas

O principal módulo do interpretador é *Query Evaluator*, responsável por obter valores finais a partir de consultas válidas. A especificação de cTVQL define um sistema de dedução formal onde se estabelece uma *relação de redução* entre consultas sob um determinado estado de banco de dados. O fato da consulta q ser reduzida para q' sob o estado $state$ é denotado $state \vdash q \rightarrow q'$. A avaliação de consultas é descrita como o fecho transitivo dessa relação. Por exemplo, uma consulta q_1 vai sendo transformada sucessivamente em $q_2 \rightarrow q_3 \rightarrow \ldots \rightarrow q_n$ até que a relação de redução esteja indefinida para um dado q_n. Vale ressaltar que todos os valores de retorno, como um conjunto de inteiros, também são considerados consultas válidas.

No caso específico de cTVQL, a avaliação de consultas não altera o estado do banco de dados e a relação de redução é uma *função parcial*. Tais características permitiram que a função de avaliação de consultas fosse implementada facilmente sob a forma de uma função $evalQuery : state \times query \rightarrow query$. Em linhas gerais, a programação do algoritmo de avaliação $evalQuery$ foi basicamente uma transcrição das regras de dedução formal para Ocaml. Essa abordagem é interessante por permitir uma implementação muito rápida, já que eventuais modificações nas semântica formal são transpostas facilmente para o interpretador. Isto, claro, desde que não se altere o caráter *funcional* da relação de redução. Outro efeito interessante é que isso diminui o risco de erros de programação, pois a implementação é muito similar à descrição formal. Em contrapartida, tal abordagem pode não ter um desempenho muito bom ao se trabalhar com grandes quantidades de dados. Tal tipo de avaliador visa primariamente a *corretude* do algoritmo em relação à especificação e uma codificação relativamente rápida, deixando questões de desempenho em segundo plano.

3.3. Verificador de tipos

Um sistema de tipos é um sistema de dedução formal que associa consultas a tipos sob um dado contexo. No caso de cTVQL, a associação da consulta q ao tipo σ sob o contexto K é denotado $K \vdash q : \sigma$. O contexto K é formado pelo esquema de classes e pelo estado do banco de dados.

Uma das atribuições de um sistema de tipos é a eliminação de programas que

levariam a situações de erro antes de sua avaliação através de uma categorização das construções de uma determinada linguagem [Pierce 2002]. Um verificador de tipos permite descartar consultas *erradas* antes da sua avaliação. Por exemplo, a consulta

```
select x.telefone
from Pessoas x
where x.nome > 10
```

é incoerente considerando o esquema apresentado na Seção 2, já que tenta comparar um atributo do tipo *string* (`x.nome`) com um literal do tipo inteiro (10).

Um consideração importante é que, em cTVQL, não se pode obter diretamente um verificador de tipos para cTVQL da mesma forma que se obteve um avaliador de consultas. Isto se dá porque em cTVQL uma mesma consulta q pode ser de vários tipos simultaneamente. Por exemplo, o valor $\{\}$, que representa um conjunto vazio, é ao mesmo tempo do tipo $Bag(Int)$ e do tipo $Bag(String)$. Essa característica do sistema de tipos constitui uma restrição importante para a definição de um verificador de tipos funcional.

A solução adotada para se obter um verificador de tipos funcional foi adicionar um tipo *indefinido* ($\perp$) à linguagem de tipos. Esse tipo $\perp$ foi colocado como subtipo de todos os outros tipos possíveis. Desta forma, o valor $\{\}$ é associado ao tipo $Bag(\perp)$ pelo verificador de tipos. As regras de avaliação de tipos foram modificadas para usar a operação de *menor limitante superior* entre tipos, e, desta forma, obter sempre um tipo mais específico. O processo *Type Checker* consiste de uma função $typeChecker : schema \times state \times query \rightarrow type$, construída a partir das regras de tipo modificadas. Tal solução, entretanto, ainda não está provadamente correta, pois é necessário que se garanta formalmente a existência de um *menor limitante superior* para todos os possíveis conjuntos de tipos associados a uma dada consulta. Tal verificação é apontada como um trabalho futuro.

3.4. Verificador de integridade do banco de dados

Como já apontado anteriormente, é necessário que o estado dos objetos esteja de acordo com o esquema de classes para que o verificador de tipos forneça associações de tipo corretas. O módulo que garante a boa formação do estado em SGBD é independente do módulo avaliador de consultas, mas foi incluído no interpretador pois consta da especificação formal de cTVQL. No interpretador cTVQL, o processo *Integrity Checker* é implementado através de uma função $validState : schema \times state \rightarrow bool$.

3.5. Ferramentas utilizadas

O interpretador foi implementado na linguagem Ocaml, utilizando-se das ferramentas `ocamllex` e `ocamlyacc` para geração de *parsers*. A interface gráfica foi construída utilizando-se a biblioteca `lablt k`. O protótipo pode ser executado em todos os sistemas operacionais para os quais foi portado o interpretador de Ocaml, incluindo Windows e Linux. Arquivos binários, código fonte e documentação estão disponíveis em `www.inf.ufrgs.br/~rma/ctvql`. Na Figura 3 é apresentado uma captura de tela do interpretador cTVQL.

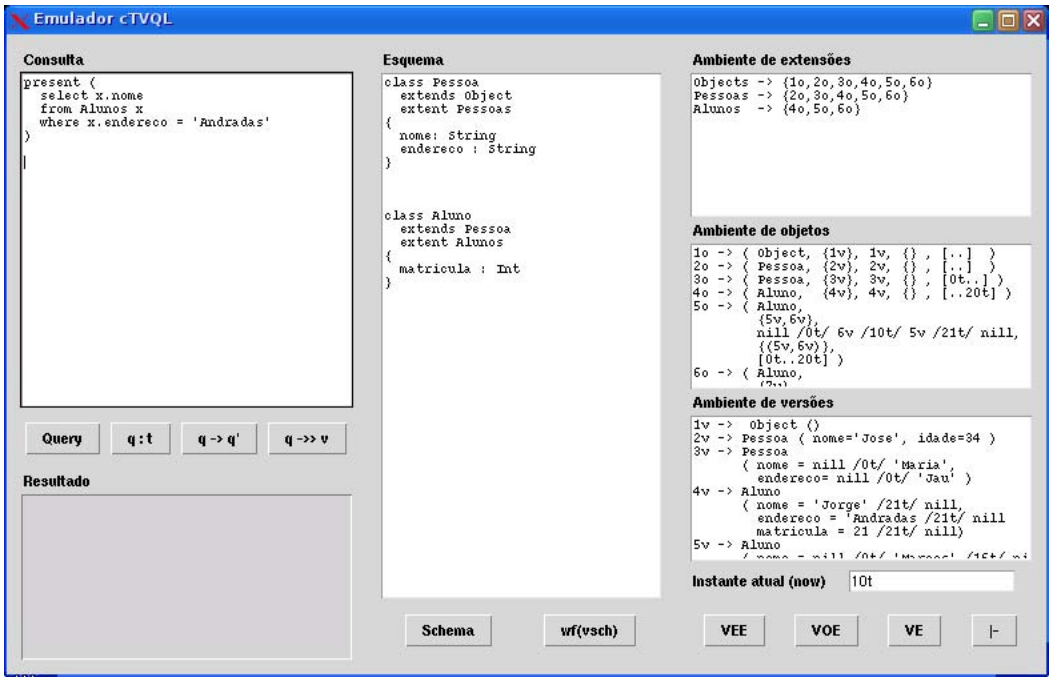

Figura 3. Captura de tela da interface do interpretador cTVQL.

4. Considerações finais

A implementação do avaliador de consultas e do verificador de tipos se deu paralelamente à formalização de TVQL, servindo como ferramenta de apoio para a melhor compreensão das construções da linguagem e para testar exemplos. A disponibilidade de ferramentas como geradores de *parsers* e o paradigma escolhido facilitaram muito a implementação dos módulos pretendidos.

Apesar desse tipo de ferramenta não ter por objetivo ambiente de produção, ele permite que sejam identificados vários problemas passíveis de serem encontrados em implementações mais robustas. Como exemplo, basta mencionar as modificações introduzidas no sistema de tipos para se obter um verificador de tipos funcional.

Referências

[Cattell et al. 2000] Cattell, R. G. G., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F., editors (2000). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann.

[Machado 2005] Machado, R. (2005). Semântica formal para TVQL. Master's thesis, Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS).

[Moro 2001] Moro, M. M. (2001). Modelo temporal de versões. Master's thesis, Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS).

[Moro et al. 2002] Moro, M. M., Edelweiss, N., Zaupa, A. P., and dos Santos, C. S. (2002). TVQL - Temporal Versioned Query Language. In *13th International Conference on Database and Expert Systems Applications*, volume 2453 of *Lecture Notes in Computer Science*, pages 618–627, Aix-en-Provence, France. Berlin: Springer-Verlag.

[Pierce 2002] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.