Criação de uma biblioteca para HasCASL

Proposta de Mestrado

Glauber Módolo Cabral – Orientando

Prof. Dr. Arnaldo Vieira Moura – Orientador

Setembro de 2007

Resumo

CASL é uma linguagem de especificação algébrica criada para ser um padrão. Através de extenções e restrições a CASL, cria-se uma família de linguagens que formam um arcabouço de especificação algébrica. A extenção que possui elementos de lógica de segunda ordem , chamada HasCASL, permite a especificação de programas funcionais com elementos muito parecidos com os presentes na linguagem de programação Haskell. Embora CASL já possua uma biblioteca de especificações prontas, HasCASL ainda carece de tal biblioteca. Dado que uma linguagem só é vastamente utilizada quando permite o reuso de códido, a presente proposta visa à criação de uma biblioteca para HasCASL baseada na biblioteca Prelude, da linguagem funcional Haskell. Prelude será tomada como base a fim de facilitar a geração de código executável a partir de uma especificação em HasCASL, já que os elementos utilizados encontrarão correspondentes na biblioteca Prelude.

1 Introdução

CASL é uma linguagem de especificação algébrica criada com o intuito de se tornar padrão. Esta linguagem permite extenções e restrições que originam uma família de linguagens de especificação, formando um arcabouço para especificação algébrica. A linguagem HetCASL (*Heterogeneous CASL*) permite a integração dos módulos escritos na demais linguagens, tornando o arcabouço capaz de suportar diversos paradigmas de programação, tais como: especificação de aspectos funcionais e reativos de sistemas; programação funcional e imperativa; lógicas de primeira e segunda ordem, modal, temporal e computacional.

Os programas escritos na família de linguagens de CASL, unidos usando-se HetCASL, podem ser formalmente provados através da ferramenta Hets. Esta ferramenta, escrita na linguagem funcional Haskell, possui conexão com os provadores de teorema SPASS e Isabelle, e trasforma as demais linguagens em códigos que podem ser provados através destes dois provadores.

Uma linguagem de especificação, assim como acontece nas linguagens de programação, precisa de bibliotecas com especificações comuns já provadas. Dessa forma, o reuso do código permite que o foco do problema seja atacado sem perda de tempo com partes comuns já realizadas em outros projetos. Embora CASL já apresente tal biblioteca, HasCASL ainda não possui uma biblioteca com elementos de lógica de segunda ordem.

A presente proposta tem o intuito de criar tal biblioteca, de forma a colaborar com o aumento no uso da linguagem HasCASL. Pretende-se partir da biblioteca Prelude, de Haskell, de forma que a biblioteca criada contenha todos os tipos definidos na Prelude, ajudando no mapeamento de especificações para códigos executáveis em Haskell.

Neste documento, a Seção 2 descreve a linguagem de especificação algébrica CASL, que é extendida e dá origem à linguagem de especificação algébrica Has-

CASL, descrita na Seção 3. A Seção 4 descreve a linguagem de programação funcional Haskell, cuja biblioteca Prelude será uma das bases do projeto. A Seção 5 descreve a linguagem HetCASL e a ferramenta Hets; esta última será utilizada pra provar formalmente a biblioteca, através do provador Isabelle, descrito na Seção 6. A Seção 7 descreve os objetivos desta proposta e o cronograma de atividades é apresentado na Seção 8.

2 CASL

A linguagem de especificação algébrica CASL surgiu como produto de uma iniciativa internacional para obter uma linguagem única de especificação que pudesse abranger o maior número de construções conhecidas. Esta seção descreve a linguagem CASL e está baseada em [Astesiano et al. 2002].

Com poucas exceções, as características de CASL estão presentes de alguma forma em outras linguagens de especificação algébrica. Porém, uma única linguagem não serve a todos os propósitos desejados: algumas características sofisticadas requerem paradigmas de programação específicos e aplicações especiais; já métodos de prototipação e geração de especificações funcionam apenas na ausência de certas características. Por exemplo, reescrita de termos requer especificações com axiomas equacionais (com quantificadores ou não).

Desta, forma, CASL foi construída de forma a ser o núcleo de uma família de linguagens: algumas sub-linguagens são obtidas através de restrições semânticas ou sintáticas, enquanto extensões são criadas para dar suporte a vários paradigmas de programação. A criação da linguagem levou em conta extensões previamente planejadas, tal qual a que suporta funções de segunda ordem. Assim, CASL está dividida em várias partes que podem ser entendidas e usadas separadamente, a saber:

- Especificações Básicas: contêm declarações (de tipos e operações), defini-

ções (de operações) e axiomas (relacionando as operações);

- Especificações Estruturais: permitem que as Especificações Básicas sejam combinadas para formar especificações maiores ;
- Especificações Arquiteturais: definem como devem ser separadas as várias especificações na implementação, de forma que o reuso de especificações dependentes entre si seja possível;
- Especificações de Bibliotecas: definem conjuntos de especificações, com construções que permitem controle de versão e de bibliotecas distribuídas pela Internet.

As Especificações Estruturais são independentes das Especificações Básicas; dessa forma, pode-se criar sub-linguagens ou extensões de CASL apenas restringindo ou extendendo as construções das Especificações Básicas, sem que seja necessário alterar os demais tipos. A seguir, descreve-se, sucintamente, as cosntruções mais importantes das Especificações Básicas.

2.1 Especificações Básicas

Uma Especificação Básica denota uma classe de modelos que são estruturas poli-sortidas parciais de primeira ordem; ou seja, álgebras poli-sortidas com funções parciais e totais e com predicados. Estes modelos são classificados por assinaturas, as quais listam: nomes de tipos, de funções (parciais e totais) e de predicados; e, também, definições (ou perfis) de funções e predicados.

As especificações possuem: declarações, que introduzem componentes da assinatura (operações, ou funções, e predicados); e axiomas, que definem propriedades sobre as estruturas que devem servir de modelos para a especificação. As operações podem ser declaradas totais (usando-se '->') ou parciais(usando-se

‘->?’), e pode-se atribuir às mesmas propriedades comuns, tais como a associatividade, evitando a necessidade de axiomatizar tais propriedades.

Operações parciais são uma maneira simples de tratar erros (como divisão por zero) e a propagação dos mesmos é direta, de tal forma que sempre que um argumento de uma operação não é definido, o resultado da operação também não é definido. Os erros e exceções também podem ser tratados por super-tipos de erros e sub-tipos. O domínio de uma função parcial pode ser definido como um sub-tipo do tipo do argumento da função, de forma a tornar esta função parcial uma função total sobre o sub-tipo. As funções podem ser declaradas totais em sua definição, ao invés de tratar a totalidade nos axiomas.

Os predicados são parecidos com operações, mas não possuem tipo de retorno; apenas o tipo dos parâmetros é declarado. Eles podem ser declarados e definidos ao mesmo tempo, ao invés de terem suas declarações e definições em seções separadas.

Os axiomas são escritos em lógica de primeira ordem sobre fórmulas atômicas. As variáveis a serem usadas nos axiomas podem ser declaradas globalmente, antes dos axiomas; ou podem ser declaradas localmente em relação a uma lista de fórmulas;ou, ainda, podem ser declaradas para cada fórmula, usando quantificação explícita.

CASL permite que sejam incluídas anotações nos axiomas, colocando-se a anotação entre ‘%(’ e ‘)%’, de forma que eles possam ser facilmente referenciados ou utilizados por ferramentas; os comentários são incluídos após ‘%%’.

As assertivas de definição são utilizadas para indicar, explicitamente, quando determinado termo está definido ou não. A definição de um termo via assertiva é equivalente a uma igualdade existencial do termo a ele mesmo. Uma igualdade existencial, declarada usando-se ‘=e=’, entre dois termos do mesmo tipo é válida quando os dois termos estão definidos e são iguais; uma igualdade

forte, declarada usando-se '=', é válida, também, quando os dois termos estão indefinidos. A interpretação das fórmulas segue a lógica de primeira ordem com dois valores (verdadeiro e falso). Existe, ainda, a assertiva de pertinência de sub-tipo, indicada por 'in', que cria um predicado relacionando a pertinência de um elemento a um tipo. Como regra, é aconselhável usar equações existenciais quando definindo condições e propriedades e usar equações fortes quando definindo funções parciais de forma indutiva.

Em CASL, usa-se semântica LOOSE para Especificações Básicas, ou seja, todas as estruturas que satisfazem os axiomas são selecionadas como modelos; esta semântica é interessante durante a análise de requisitos, de forma que estes últimos sejam abrangentes.

Pode-se declarar um tipo de dado como livre, alterando a semântica LOOSE para a semântica inicial. Desta forma, valores de um mesmo tipo que diferem apenas na sequência da aplicação dos seus construtores são tratados como elementos diferentes deste tipo.

A terceira semântica permitida em CASL força os tipos de dados a serem gerados apenas pelos seus construtores. Isto elimina a confusão entre termos, ou seja, a menos que axiomas forcem uma igualdade, os termos são todos diferentes entre si. Esta reintrodução de confusão entre termos é possível através de axiomas.

As declarações de tipos são as únicas em que a visibilidade não é linear. Em todo o restante das especificações, um tempo só pode ser utilizado depois de ser declarado.

3 HasCASL

Esta seção apresenta a linguagem HasCASL baseando-se em [Schröder e Mossakowski]. A definição formal da linguagem encontra-se em [Schröder, Mossakowski e Maeder 2003].

A linguagem HasCASL é uma extensão de CASL, anteriormente descrita, com conceitos de lógica de segunda ordem tais como tipos que são funções, polimorfismo e construtores de tipos. HasCASL foi criada de forma que a linguagem de programação funcional Haskell seja seu sub-conjunto; dessa forma, torna-se possível transformar uma especificação em HasCASL em um programa Haskell de uma forma mais simples. Como citado anteriormente, para extender CASL é necessario, apenas, extender ou alterar a Especificação Básica, sem que seja necessário alterar os outros três tipos de especificação.

A lógica de segunda ordem tradicional não permite os tipos e funções recursivas amplamente usados em linguagem funcional. HasCASL tenta resolver o problema sem utilizar semântica denotacional, fazendo emergir uma lógica interna às λ-abstrações sem que esta lógica seja um conceito primitivo da linguagem. Dessa forma, mantém-se a linguagem próxima a CASL e, ao mesmo tempo, com as propriedades desejadas de lógica de segunda ordem.

As sentenças em HasCASL diferem de CASL em dois aspectos:

- Os quantificadores (universal, existencial e unicamente existencial) podem ser aplicados sobre as variáveis de tipo e possuem restrições de sub-tipos;
- Os predicados de CASL são substituídos por termos do tipo *unit*.

Diferentemente das linguagens de programação funcional, os operações polimórficos precisam ser explicitamente instanciados, uma vez que ainda não está claro, teoricamente, como se relacionam a resolução de sobrecarga de sub-tipos e a instanciação implícita.

Como HasCASL tenta manter-se o mais próximo de CASL, sua semântica também é baseada em teoria dos conjuntos. Para que as assinaturas de segunda ordem sejam modeladas nesta semântica, é usada a noção de modelo Henkin intensional. Neste modelo, os tipos de funções são interpretados por conjuntos arbitrários equipados com uma operação de aplicação do tipo apropriado (em

oposição a um tipo parcial $s- >?t$ ser interpretado pelo conjunto completo de todas as funções parciais de s em t). A interpretação dos λ-termos é parte da estrutura do modelo ao invés de ser apenas um axioma existencial.

O modelo intensional tem algumas vantagens, entre as quais: eliminar os problemas de completude; permitir modelos iniciais de assinaturas que contenham funções parciais; e permitir que a semântica operacional das linguagens de programação funcional seja aplicada, ao invés de usar diretamente a lógica de segunda ordem. Ao contrário de Haskell, na qual a avaliação de funções é preguiçosa, a avaliação de funções em HasCASL é estrita, ou seja, argumentos indefinidos sempre resultam em valores indefinidos. Uma maneira de emular a avaliação preguiçosa, na qual é permitido que funções deixem de avaliar argumentos não utilizados internamente a fim de retornar valores definidos mesmo na presença de argumentos indefinidos, é mover os parâmetros para tipos unitários ($unit- >?a$).

Os tipos são definidos pelo uso da palavra reservada *type*, que pode ser precedida pelos qualificadores $free$ e *generated*, como em CASL. Definições de tipo que possuam tipos de funções como argumentos de construtores e recursão apenas no lado direito da seta devem ser realizadas com a palavra-reservada $cofree$; com recursão em ambos os lados da seta, os tipos devem ser definidos com $free$.

Funções recursivas, por sua vez, são definidas utilizando a abreviatura $op\ f : t = rec\,\alpha$ que equivale a declarar a operação $op\ f : t$ e adicionar o axioma $f = Y(\lambda f : t.a)$, onde Y é o operador de ponto fixo.

4 Haskell

Esta seção apresenta algums elementos sintáticos da linguagem Haskell. As informações aqui fornecidas, bem como conceitos mais aprofundados, podem

ser obtidos em materiais encontrados na internet [Haskell Team 2007] ou em livros [Thompson 1999].

Haskell é uma linguagem de programação funcional pura, fortemente tipificada e com avaliação preguiçosa (*lazy evaluation*) que surgiu da necessidade de padronização no campo das linguagens funcionais. A linguagem é funcional porque implementa os conceitos do λ-Cálculo; logo, a programação é feita através de aplicações de funções ou computações. A linguagem é fortemente tipificada, ou seja, os tipos de funções e valores precisam ser definidos explicitamente ou o compilador irá tentar resolver o tipo para o mais abrangente possível no contexto atual.

Os conceitos de avaliação preguiçosa e avaliação estrita (*strict evaluation*) estão relacionados com a interpretação dos parâmetros de uma função. Uma linguagem com avaliação estrita avalia todos os parâmetros da chamada de uma função antes de executar o corpo da função chamada. No caso de Haskell, os parâmetros de uma função são avaliados apenas quando tornam-se necessários para a execução da função.

A linguagem é pura porque não permite que a aplicação de uma função altere o estado do sistema (a menos das partes envolvidas nos cálculos do qual a aplicação depende). Algumas linguagens funcionais são ditas não-puras porque permitem que, além de efetuar a aplicação de funções, um programa defina ações tipicamente de linguagens imperativas, como exibir algo em tela ou salvar um conteúdo em um arquivo. Estas ações causam efeitos colaterais, alterando o mundo real.

O mundo real pode ser entendido como tudo que está diretamente ligado ao sistema em que o programa está rodando; os efeitos colaterais, por sua vez, são todos os efeitos que não sejam o cálculo de funções. Haskell permite executar ações que alterem o mundo real através de uma entidade matemática chamada

mônada. As mônadas seqüenciam computações passando, implicitamente, o estado atual do mundo real; isto permite que os efeitos colaterais não interfiram no código do programa.

Pode-se definir uma função, em Haskell, diretamente como se faria em λ-Cálculo, com tipos ou não. Uma vantagem de se utilizar uma linguagem de programação é poder reutilizar métodos definidos através de nomes dados aos mesmos. Nomeia-se, então, uma função cujo corpo está escrito em λ-Cálculo a fim de reutilizá-la. Como o tipo dos elementos não foi declarado, o compilador irá calcular os tipos de acordo com o contexto. Para definir-se o tipo de uma função, deve-se escrever o nome da mesma seguido por "::" e pela seqüência de tipos de cada parâmetro, separados por "->", com associação à direita.

Dessa forma, sem se afastar muito do λ-Cálculo, é possível escrever programas em Haskell. Contudo, na maioria dos casos, as funções são definidas de acordo com a sintaxe de Haskell, por ser mais prática e fácil. Uma função é definida por um nome seguido por uma seqüência de parâmetros separados do corpo da função pelo sinal de "=".

Faz-se necessário diferenciar uma função de um operador. Uma função, em Haskell, é sempre definida de forma prefixa; um operador tem sua definição infixa. Uma função prefixa pode ser usada de forma infixa colocando-a entre crases; um operador, por sua fez, pode ser utilizado de forma prefixa colocando-o entre parênteses.

Outra característica explorada é a coincidência de padrões (*pattern matching*). Uma função para calcular o fatorial de um inteiro pode ser definida, utilizando-se coincidência de padrões, como se segue:

```
fat :: Int-> Int
fat 1 = 1
fat x = x * fat(x-1)
```

Cada chamada à função *fat* percorrerá, em ordem, as duas definições da função. Se o parâmetro for o inteiro 1, ele será retornado; caso contrário, será feita uma chamada recursiva à função com o parâmetro decrescido de uma unidade.

Haskell permite a criação de bibliotecas, que são chamadas de módulos e são criadas com a palavra-chave *module*. A linguagem possui uma biblioteca padrão chamada *Prelude*, na qual são definidas funções básicas que operam sobre os tipos primitivos: *Int*, *Bool*, *Char*, *Float*; e sobre uplas (seqüências de valores entre parênteses) e listas definidas sobre os tipos primitivos.

Haskell, como as demais linguagens funcionais, trabalha com listas como ferramenta principal para representação de dados. Pode-se criar listas de tipos primitivos, listas de uplas, listas de listas, listas de funções; todos os elementos da lista, contudo, precisam ser de tipos idênticos. A ordem e a quantidade de elementos dentro de uma lista também são importantes para efeito de igualdade. A lista $[1, 2]$ não é igual à lista $[2, 1]$ e ambas são diferentes das lista $[1, 1, 2]$.

Dois operadores básicos para se manipular listas são o operador ":" (construção) e o operador "++" (concatenação). Uma lista sempre é construída a partir de uma lista vazia unindo-se a esta o elemento desejado através do construtor de listas; dessa forma, as seguintes listas são eqüivalentes: $[1, 2, 3] = 1 : 2 : 3 : [] = 1 : 2 : [3] = 1 : [2, 3$. Para que se possa concatenar duas listas, elas precisam ser do mesmo tipo. As listas $[1, 2, 3] :: [Int]$ e $[4, 5, 6] :: [Int]$ podem ser concatenadas por $[1, 2, 3] + +[4, 5, 6]$, produzindo a lista $[1, 2, 3, 4, 5, 6]$.

Uma ferramenta fundamental da linguagem Haskell é a definição de Tipos de Dados. Define-se um tipo com a palavra-chave *data* seguida do nome do tipo e do sinal de igual; em seguida, listam-se os elementos que compõem o tipo (os construtores), separados por "|". Tanto o nome do tipo quanto dos construtores devem iniciar-se com letra maiúscula. Um construtor pode ser apenas um nome ou um nome seguido de tipos ou variáveis de tipos; as variáveis servem para

definir tipos polimorfos e devem ser unidas, através de parênteses, ao nome do construtor, de forma a evitar ambigüidade.

Por fim, pode-se incluir a especificação de um tipo de dado em uma biblioteca. A declaração deve exportar, em sua definição, todos os métodos definidos pela especificação, garantindo a integridade do tipo de dado e as propriedades provadas para o tipo em questão. Os nomes devem ser exportados no formato *módulo.operação*, de forma que não gere conflitos de nomes com variáveis de outros pacotes presentes no escopo atual. Isto é uma limitação da linguagem que já está sendo aprimorada pelo grupo de trabalho do compilador *GHC* [GHC Team 2007].

5 Especificações Heterogêneas: HetCASL e Hets

Na área de métodos formais e lógica são utilizadas diversas lógicas distintas porque, atualmente, não se conhece uma maneira de combiná-las de forma a obter uma única lógica que se preste a todos os usos desejados. Especificar sistemas grandes, então, exige o uso de especificações heterogêneas porque determinados aspectos são melhor abordados por uma lógica específica. Uma especificação heterogênea possui uma interoperabilidade formal entre as linguagens, de forma que cada linguagem possua seu método de prova e que todas as provas sejam consistentes quando vistas do ponto de vista da especificação heterogênea.

As diversas sub-linguagens e extensões de CASL são únidas pela linguagem "Heterogeneous CASL (HetCASL)"[Mossakowski Christian Maeder e Wölfl], que possui as construções estruturais de CASL. HetCASL estende a propriedade da semântica de CASL ser independente de instituição com construções que indicam a relação entre a tradução entre especificações que ocorre em conjunto com a tradução entre lógicas. Vale ressaltar que "HetCASL" preserva o fato de as lógicas individualmente usadas pelas especificações são ortogonais a CASL.

A ferramenta "Heterogeneous Tool Set (Hets)"[Mossakowski Christian Maeder e Wölfl] é um analisador sintático e um gerenciador de provas para HetCASL, implementado em Haskell, que combina as várias ferramentas de provas existentes para cada uma das lógicas individuais utilizadas nas diversas sub-linguagens e extensões de CASL. Hets baseia-se em um grafo de lógicas e linguagens, provendo uma semântica clara para especificações heterogêneas bem como um cálculo de provas.

Cada lógica do grafo é representada por um conjunto de tipos e funções em Haskell. A sintaxe e a semânticas das especificações heterogêneas em HetCASL e as suas respectivas implementações são parametrizadas por um grafo de lógicas arbitrário dentro da ferramenta Hets. Isto permite que a implementação da ferramenta e dos módulos para cada lógica sejam facilmente gerenciáveis do ponto de vista de engenharia de software.

Entre as diversas lógicas suportadas por Hets, são de interesse: HasCASL, Haskell, SPASS e Isabelle. As dúas últimas são as únicas que possuem provadores, a saber: SPASS [Weidenbach et al. 2002] é um provador automático de teoremas para lógicas de primeira ordem com igualdade; Isabelle [Nipkow, Paulson e Wenzel 2002] é um provador semi-automático de teoremas para lógicas de segunda ordem. As demais lógicas são provadas através de traduções entre elas mesmas e uma das duas lógicas com provadores.

A estrutura de provas em Hets baseia-se no formalismo de grafos de desenvolvimento [Mossakowski, Autexier e Hutter 2006], amplamente utilizado para especificações de sistemas industriais. A estrutura de grafos permite a visualização direta da estrutura da especificação e facilita o gerenciamento de especificações com muitas sub-especificações.

Um grafo de desenvolvimento consite em um conjunto de nós (correspondentes a especificações completas ou trechos de especificações) e um conjunto

de arcos chamados arcos de definição que indicam a dependencia entre as várias espeficicações e suas sub-especificações. A cada nó estão associados uma assinatura e um conjunto de axiomas locais, sendo que estes últimos são herdados pelos nós dependentes através dos arcos de definição. Diferentes tipos de arcos são utilizados para indicar quando há, ou não, mudança da lógica entre dois nós.

Um segundo tipo de arco, os arcos de teorema, são utilizados para postular relações entre diferentes teorias, servindo para representar as necessidades de prova que surgem durante o desenvolvimento. Os arcos de teorema podem ser globais ou locais (representados por arcos de formas distintas no grafo): os arcos globais indicam que todos os axiomas do nó fonte são válidos no nó alvo; os arcos locais indicam que apenas os axiomas do nó fonte são válidos no nó alvo.

Os arcos de teoria globais são decompostos em arcos mais simples (globais ou locais) atrávés do cálculo de prova para gafos de desenvolvimento. Os arcos locais podem ser provados tranformando-os em alvos locais de prova, os quais podem ser provados utilizando o cálculo de prova da lógica representada no nó.

Abaixo estão reproduzidos dois trechos de códigos utilizados pela ferramentas Hets. O primeiro, escrito na lógica CASL, extende um tipo previamente definido (AllenHayes), incluindo definições e sentenças; estas últimas decorrem das primeiras e geram obrigatoriedade de prova para as sentenças:

```
logic CASL
spec ConstructPointsFO[AllenHayes] = %def
  preds __ __ Equi __ __, __ __ Less __ __: Elem x Elem x Elem x Elem
  forall x,y,z,u:Elem
  . x y Equi z u <=> z M y /\ z M u /\ z M u
  . x y Less z u <=> x M y /\ z M u /\ (exists v:Elem . x M v /\ v M u)
then %implies
```

```
  forall x,y,z,u:Elem
  . x M y => x y Equi x y
  . x y Equi z u => z u Equi x y
```

O segundo exemplo, escrito na lógica HasCASL, especifica uma interface, declarando um predicado visível nesta interface e definindo seu funcionamento através de uma sentença:

```
logic HasCASL
view FlowOfTime_In_ConstructPointsFromIntervals[AllenHayes]:
  FlowOfTime to
  { ConstructPointsFromIntervals[AllenHayes] then %def
    pred __ < __ : Inst X Inst
    forall X,Y:Inst . X < Y <=> exists x,y,u,v:Elem
      . x M y /\ u M v /\ X = eqcl(x,y)/\ Y = eqcl(u,v)
        /\ x y Less u v}
= sort Elem |-> Inst
```

O código do qual os trechos foram retirados origina o grafo de desenvolvimento da Figura 1.

6 Isabelle

Esta seção descreve o provador de teoremas Isabelle baseando-se na documentação presente em seu *site* na Internet [Comunity 2007]; uma descrição completa pode ser encontrada no manual do provador [Nipkow, Paulson e Wenzel 2002].

Isabelle é um provador de teoremas genérico que permite o uso de várias lógicas como cálculo formal na prova de teoremas. Hets utiliza Isabelle para provar teoremas em lógica de segunda ordem, mas o provador permite , por exemplo, o uso de teoria de conjuntos axiomatizada, dentre outras lógicas. O

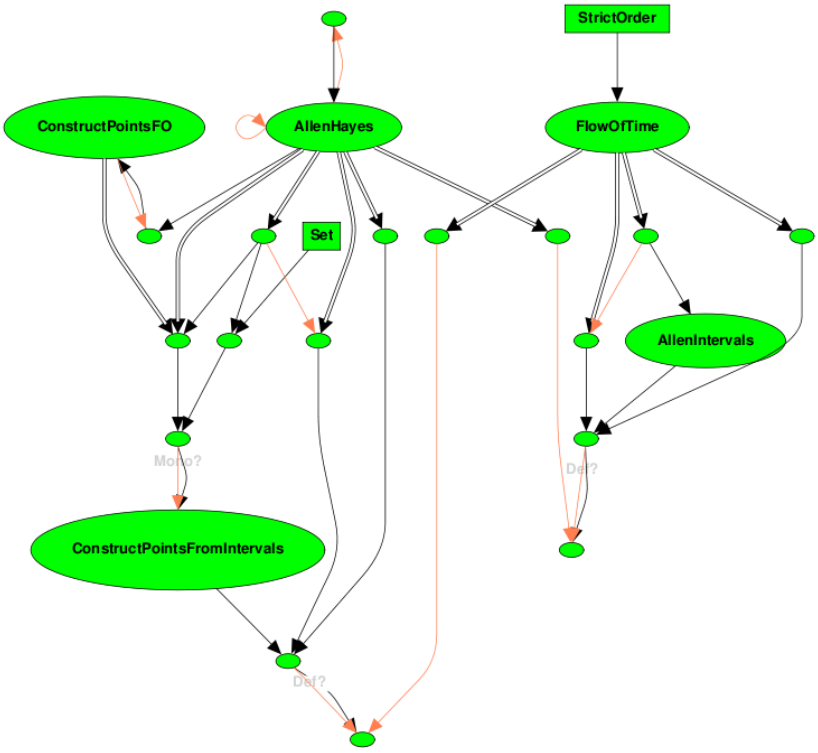

Figura 1: Exemplo de grafo de desenvolvimento

suporte a várias lógicas é uma das características de destaque da ferramenta.

O provador possui um excelente suporte à notação matemática: novas notações podem ser incluídas utilizando-se símbolos matemáticos comuns e as provas podem ser descritas de forma estruturada ou, mais facilmente, como seqüência de comandos. As definições e provas podem incluir códigos na linguagem de formatação TeX, de forma que documentos formatados possam ser gerados diretamente do código fonte.

A maior limitação de provadores de teorema é a necessidade de grande esforço e experiência por parte dos usuários para que as provas sejam realizadas. Visando a facilitar este processo, Isabelle possui algumas ferramentas que automatizam alguns trechos de provas, tais como equações, aritmética básica e

fórmulas matemáticas.

A lógica de segunda ordem HOL é utilizada na codificação das teorias. Sua sintaxe assemelha-se muito às linguagens de programação funcional por ser baseada em cálculo lambda tipificado. Esta lógica permite construções de tipos de dados, tipos de funções e outras construções comuns em linguagens funcionais. A traadução de HasCASL para HOL é feita pela ferramenta Hets de forma automática.

Isabelle possui uma extensão, chamada Isar, que permite descrever as provas de forma que tanto humanos possam lê-las como computadores possam interpretá-las facilmente. O provador possui uma extensa biblioteca de teorias matemáticas já provadas (por exemplo, nas áreas de Álgebra e Teoria de Conjuntos) e, também, muitos exemplos de provas realizadas em Verificação Formal.

A seguir, apresenta-se a codificação em Isar do Teorema de Cantor, que afirma a existência de mais subconjuntos do que elementos em um conjunto qualquer. O código descreve como a prova deve ser realizada, porém, poderia ter sido escrito de forma que Isabelle efetuasse uma prova automática.

```
theory Cantor imports Main begin


theorem "EX S. S ~: range (f :: 'a => 'a set)"
proof
  let ?S = "{x. x ~: f x}"
  show "?S ~: range f"
  proof
    assume "?S : range f"
    then obtain y where "?S = f y" ..
    then show False
    proof (rule equalityCE)
```

```
      assume "y : f y"
      assume "y : ?S" then have "y ~: f y" ..
      with ‘y : f y‘ show ?thesis by contradiction
    next
      assume "y ~: ?S"
      assume "y ~: f y" then have "y : ?S" ..
      with ‘y ~: ?S‘ show ?thesis by contradiction
    qed
  qed
qed
```

7 Proposta

Um pré-requisito para o uso prático de uma linguagem de especificação é a disponibilidade de um conjunto de especificações padrões previamente disponíveis [Schröder 2006]. A linguagem CASL possui tal conjunto descrito em "CASL Basic Datatypes" [Roggenbach, Mossakowski e Schröder 2004]; o documento, ao invés de fornecer blocos comuns para reuso como acontece com bibliotecas em linguagens de programação, fornece exemplos completos de especificações que ilustram o uso de CASL tanto no nível das Especificações Básicas quanto das Especificações Estruturais. As especificações incluídas são tanto de tipos básicos de dados como de especificações que expressam propriedades de estruturas. No primeiro caso, estão presentes tipos de dados simples, como números e caracteres, bem como tipos de dados estruturados, como listas, vetores e matrizes; no segundo caso, estão incluídos estruturas algébricas tais como monóides e anéis e propriedades matemáticas como relações de equivalência e de ordem parcial.

Atualmente, a linguagem HasCASL não possui uma biblioteca nos moldes

da biblioteca de CASL. Segundo Scröder, os tipos de dados descritos em "CASL Basic Datatypes" podem servir como base para a construção de uma biblioteca padrão paras as extensões de CASL. No caso de HasCASL, sugere-se a inclusão de novas especificações que envolvam características de ordem superior tais como a completude de ordenações parciais assim como estender os tipos de dados efetuando, antes, a troca da parametrização no nível das Especificações Estruturais pela dependência de tipos. Um exemplo sugerido é a inclusão de funções de ordem superior que operem sobre listas, tais como as funções *map*, *filter* e *fold*, utilizando-se especificações heterogêneas (usando-se CASL e HasCASL) para maximizar o reuso.

Com base nestas sugestões, propõe-se iniciar a criação de uma biblioteca para HasCASL tendo como base o documento citado e a biblioteca Prelude, de Haskell. Criar tal biblioteca mostra-se uma tarefa útil, dado que contribuirá para o aumento de uso da linguagem. Como a biblioteca Prelude é uma biblioteca padrão que deve ser disponibilizada por todo compilador Haskell, transportá-la para HasCASL contribuirá para que as especificações sejam mais facilmente mapeadas para código executável em Haskell.

A criação de tal biblioteca exige o estudo da teoria de linguagens de programação funcional no tocante aos tipos e funções de ordem superior e a busca de soluções para a inclusão de tais elementos na biblioteca de forma a não alterar a bliblioteca de CASL mas, sim, estendê-la.

Os códigos gerados serão verificados através da ferramenta Hets, gerando provas formais para os mesmos de forma a garantir a corretude de todo o código gerado. Como dito anteriormente, esta ferramenta gerencia as provas e utiliza dois provadores para executar as provas. Faz-se necessário, então, aprender a utilizar os dois provadores (SPASS e Isabelle).

Isabelle é o provador que demanda maior trabalho. Sua utilização será maior

que a do provador SPASS, uma vez que os códigos que se pretende implementar envolvem os conceitos de ordem superior que são provados através do Isabelle. Como este é um provador semi-automático, será necessária intervenção manual na prova; isto demanda um aprendizado profundo do sistema, desde a sintaxe utilizada para os arquivos de entrada até as saídas geradas pelo provador. O conhecimento dos possíveis erros gerados torna-se fundamental para que possíveis erros dos códigos possam ser procurados e corrigidos.

Uma posível extensão do trabalho, a ser considerada durante o decorrer da codificação, é a especificação de alguns outros tipos de dados relacionados a Teoria das Categorias. Isto seria útil na medida em que a base teórica por trás de CASL e de HasCASL está fortemente ligada a esta teoria. Dessa forma, pode-se iniciar o estudo teórico deste tópico com vistas ao início do Doutorado.

8 Cronograma

A seguir, apresentam-se as atividades a serem realizadas durante o projeto de mestrado. A Tabela 1 possui o tempo estimado das atividades:

1. Créditos obrigatórios do mestrado;
2. Estudo das linguagens HasCASL e CASL, com ênfase nas construções de HasCASL e na biblioteca de CASL;
3. Estudo da ferramenta Hets e do provador Isabelle;
4. Exame de Qualificação do Mestrado;
5. Estudo da biblioteca Prelude e dos conceitos de lógica de segunda ordem envolvidos na bilbioteca;
6. Implementação e prova de corretude da biblioteca para HasCASL.

7. Escrita de um artigo para congresso.

8. Escrita da dissertação.

9. Defesa e revisão.

Atividade	2007					2008						2009
	3-4	5-6	7-8	9-10	11-12	1-2	3-4	5-6	7-8	9-10	11-12	1-2
1	•	•	•	•	•							
2		•	•									
3				•	•							
4				•								
5						•	•					
6								•	•	•		
7										•		
8			•		•		•				•	
9												•

Tabela 1: Cronograma de Atividades: março/2007 até fevereiro/2009

Referências

[Astesiano et al. 2002]ASTESIANO, E. et al. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 2002. Disponível em: <http://citeseer.ist.psu.edu/astesiano01casl.html>.

[Comunity 2007]COMUNITY, I. *Isabelle Overview*. 2007. Disponível em: <http://isabelle.in.tum.de/overview.html>.

[GHC Team 2007]GHC Team, T. *The Glasgow Haskell Compiler*. 2007. Disponível em: <http://haskell.org/ghc/>.

[Haskell Team 2007]Haskell Team, T. *Learning Haskell*. 2007. Disponível em: <http://www.haskell.org/haskellwiki/Learning_Haskell>.

[Mossakowski Christian Maeder e Wölfl]MOSSAKOWSKI CHRISTIAN MAEDER, K. L. T.; WöLFL, S. *The Heterogeneous Tool Set*. Disponível em: <http://www.informatik.uni-bremen.de/ till/papers/hets-paper.pdf>.

[Mossakowski, Autexier e Hutter 2006]MOSSAKOWSKI, T.; AUTEXIER, S.; HUTTER, D. *Development graphs - Proof management for structured specifications*. v. 67, n. 1-2, 2006. 114-145 p.

[Nipkow, Paulson e Wenzel 2002]NIPKOW, T.; PAULSON, L. C.; WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. [S.l.]: Springer, 2002. (LNCS, v. 2283).

[Roggenbach, Mossakowski e Schröder 2004]ROGGENBACH, M.; MOSSAKOWSKI, T.; SCHRÖDER, L. CASL libraries. In: CASL *Reference Manual*. [S.l.]: Springer, 2004, (LNCS Vol. 2960 (IFIP Series)). part V.

[Schröder 2006]SCHRÖDER, L. Higher order and reactive algebraic specification and development. Feb 2006. Disponível em: <http://www.informatik.uni-bremen.de/ lschrode/papers/Summary.ps>.

[Schröder e Mossakowski]SCHRÖDER, L.; MOSSAKOWSKI, T. *HasCASL: Towards Integrated Specification And Development Of Haskell Programs*. Disponível em: <`http://citeseer.ist.psu.edu/459463.html`>.

[Schröder, Mossakowski e Maeder 2003]SCHRÖDER, L.; MOSSAKOWSKI, T.; MAEDER, C. *HasCASL - Integrated functional specification and programming. Language summary*. 2003.

[Thompson 1999]THOMPSON, S. *Haskell: The Craft Of Functional Programming*. 2. ed. Boston, USA: Addison Wesley, 1999. 512 p. ISBN 0-201-34275-8.

[Weidenbach et al. 2002]WEIDENBACH, C. et al. SPASS version 2.0. In: VORONKOV, A. (Ed.). *Proceedings of the 18th International Conference on Automated Deduction*. [S.l.]: Springer-Verlag, 2002. (LNAI, v. 2392), p. 275–279.