Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Formalização da Terminação de Especificações Funcionais

Thiago Mendonça Ferreira Ramos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Mauricio Ayala Rincón

Brasília
2017

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Formalização da Terminação de Especificações Funcionais

Thiago Mendonça Ferreira Ramos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Mauricio Ayala Rincón (Orientador)
CIC/UnB

Prof. Dr. Edward Hermann Haeusler
PUC-Rio

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 02 de março de 2017

Dedicatória

Dedico essa dissertação aos meus pais, Edson Sergio Ferreira Ramos e Ana Lucia Mendonça Ferreira Ramos, por todo carinho e cuidados de todos esses anos.

Agradecimentos

Agradeço a meu orientador, Mauricio Ayala Rincón, por me auxiliar no dia a dia para concluir essa dissertação. Agradeço também aos meus colegas de laboratório pelo apoio que me deram. Gostaria de agradecer pela intersessão de São Pio de Pietrelcina e pela paciência com a insistência das minhas orações e gostaria de agradecer a Deus pela oportunidade de concluir o mestrado.

Resumo

Terminação é uma propriedade crítica para formalização de correção de programas. Verificar automaticamente terminação de um programa é conhecido como Problema da Parada e Turing provou que é um problema indecidível. Apesar disso, é possível construir algoritmos de semi decisão para verificar terminação, que respondem 'sim' se pode provar que o algoritmo para e 'não sei' caso contrário. Para construir esses algoritmos de semi decisão é necessário considerar diferentes noções de terminação, provando que são equivalentes. Neste trabalho, noções de terminação são formalizadas equivalentes para uma linguagem funcional de primeira ordem chamada PVS0 usando o assistente de prova *Prototype Verification System*. Essas noções são: as funções produzem uma saída, a árvore de derivação de chamados recursivos de funções tem tamanho finito (ambas as noções são chamadas terminação semântica), e os argumentos das funções decrescem para cada chamado recursivo (essa noção é chamada *ranking function*). As contribuições desse trabalho incluem a formalização de alguns lemas necessários para demonstrar equivalência entre noções de terminação semântica e *ranking function*, e como resultado principal a formalizações de indecidibilidade do Problema da Parada e Turing-Completude de PVS0.

Palavras-chave: Correção de Programas, Automação da Terminação, Indecidibiliade do Problema da Parada, Turing Completude

Abstract

Termination is a critical property for the formalization of programs correctness. Verifing automatically termination of a program for an input is known as Halting Problem and Turing proved that this is undecidable. However, it is possible to build semi decision algorithms for the verification of termination, that answer 'yes' if it is possible to prove that the algorithm halts, and 'do not know' otherwise. To construct these semi decision algorithms it is necessary to consider different notions of termination, proving that they are equivalent. In this work, notions of termination were formalized equivalent for a minimal functional first order language called PVS0 using the proof assistant Prototype Verification System. These notions are: the functions produces an output, the derivation tree of recursive calls of functions has a finite size (both these notions are called semantic termination), and the arguments of functions decreases for each recursive call (this notion is called ranking function). The contributions of this work includes formalization of lemma related with the equivalence between notions of semantic and ranking function termination,and the main results are the formalization of indecidability of Halting Problem and Turing-Completeness of PVS0.

Keywords: Correction of Programs, Automation of Termination, Undecidability of the Halting Problem, Turing-Completeness

Sumário

Capítulo 1

Introdução

Terminação de programas é um problema fundamental em ciência da computação, pois a prova de correção de algoritmos depende de se assegurar que os programas param com determinado tipo de entrada. Um exemplo é o problema de terminação de *threads* de *drivers* de teclado em sistemas operacionais [7]. Se após apertar uma tecla uma *thread* não termina, perde-se o controle do teclado. Entretanto, determinar se um programa para é um problema indecidível (conhecido como *halting problem* [21]) e, portanto, não é possível construir um compilador que para todo tipo de programa verifique a sua parada.

Apesar de ser um problema indecidível, pode-se construir um algoritmo de semi decisão que, dado um programa, responde "para" ou "não sei". Para isso, são usadas definições já conhecidas de terminação. Nesse sentido, existe um torneio para analisar novas técnicas, que ano após ano são melhoradas, o chamado *Annual International Workshop on Termination and Termination Competition*[1]. Ano após ano as técnicas de verificação de terminação são melhoradas de tal forma que se responde cada vez menos "não sei" [8].

Para apoiar isso, é preciso formalizar técnicas existentes de prova de terminação, para se tentar ter total segurança de correção de funcionamento. As definições existentes são, para todas as entradas:

- a função produz uma resposta;
- a árvore de chamadas recursivas possui tamanho finito;
- há um decrescimento sobre uma ordem bem fundada (uma ordem na qual para qualquer elemento, existe um elemento menor ou igual que é mínimo) entre os argumentos e os argumentos chamados na recursão (*ranking function*, [22]);
- toda sequência de chamadas recursivas causa um decrescimento finito de alguns dos argumentos (princípio da mudança de tamanho [16]). Como aplicação disso, tem-se:

[1] `http://www.termination-portal.org`

– há diminuição de argumentos no grafo cujos vértices representam as entradas da função e as arestas representam as possíveis chamadas (grafos de contexto de chamado, [17]);

– no grafo de matrizes com pesos de medida construído a partir da função produz matrizes positivas nas operações das arestas (terminação por matrizes com peso de medida [4]), o que indica decréscimo dos argumentos da função;

Aqui apenas as três primeiras definições serão focadas.

Todas essas três definições de terminação foram formalizadas no assistente de prova *Prototype Verification System* (PVS) em parceria entre membros do Grupo de Teoria da Computação da UnB e do grupo de métodos formais da NASA LaRC e NIA, para uma linguagem funcional de primeira ordem mínimal PVS0. Foram formallizadas 269 propriedades de uma biblioteca PVS0 e 348 propriedades de uma biblioteca com grafos com matrizes de medidas (*matrix weighted graphs* ou MWG) e com grafos de contexto de chamado (*calling context graphs* ou CCG) que é a biblioteca CCG . Nesta dissertação, foi formalizada a indecidibilidade do Problema da Parada da linguagem, ou seja, não existe função PVS0 que expresse se outra função PVS0 termina ou não. Também foi formalizada a Turing-Completude de PVS0, ou seja, PVS0 tem o poder expressivo das Máquinas de Turing ou de uma linguagem de programação. Entretanto precisou-se assumir que existe uma função sobrejetiva entre o tipo de PVS0 e expressões em PVS0, pois é preciso garantir que as expressões PVS0 tenham uma representatividade no tipo de PVS0 para que se tente verificar parada. Como expressões PVS0 são enumeráveis, caso não exista essa função o tipo de PVS0 tem que ser finito. Nesse caso PVS0 funcionaria como uma Máquina de Turing com fita finita, onde o problema da parada é decidível. Um aspecto relevante da prova de Turing-completude é o fato da linguagem PVS0 não apresentar mecanismos naturais para construção da composição de funções especificadas nesta linguagem, de maneira que é necessária uma prova existencial a diferença do que ocorre em modelos computacionais como Máquinas de Turing, lógica combinatória, máquinas com registros, e, em geral, linguagens mais elaboradas. As formalizações de indecidibilidade do Problema da Parada e Turing-Completude de PVS0 foram apresentadas e publicadas em [18]. Os arquivos das bibliotecas PVS0 e CCG estão representados pela figura 1.1.

Os arquivos de PVS0 foram agrupados em três categorias nessa figura: noções de terminação, que contém a formalização das equivalências entre as noções de terminação, Problema da Parada e Turing-Completude de PVS0 e exemplos de uso da teoria. Os dez arquivos em oliva são arquivos trabalhados nesta dissertação e as setas indicam que arquivos foram importados. As setas em cinza indicam que arquivos da biblioteca CCG foram importados por arquivos da biblioteca PVS0.

A descrição dos dez arguivos trabalhados é:

- pvs0_undecidability: contém a formalização do teorema de indecidilidade do Problema da Parada para PVS0.
- pvs0_comp: contém a formalização de Turing Completude de PVS0.
- measure_termination (m_ termination): contém o teorema de equivalência entre *ranking function* e terminação semântica.
- measure_termination_defs (m_termination_defs): contém a definição de *ranking function*.
- pvs0_props: contém lemas auxiliares para o teorema de equivalência entre as noções de terminação.
- pvs0_cc: contém definições auxiliares usadas em *ranking function*
- gcd_pvs0, gcd_pvs, ack_pvs0 e ack_pvs: contém exemplos de uso da teoria para provar terminação de funções, como gcd e Ackermann.

Outros treze arquivos da biblioteca PVS0 são:

- pvs0_to_ccg e ccg_to_pvs0: contém a equivalência entre terminação via o método de grafos de contexto de chamado e terminação semântica.
- scp_iff_pvs0: contém a equivalência entre terminação via princípio da mudança de tamanho e terminação semântica.
- pvs0_expr: contém a equivalência entre as noções de terminação semântica.
- pvs0_expressibility: contém que terminação pode ser expressa via linguagem PVS0.
- pvs0_pvs_T: contém a formalização de que uma função PVS pode ser expressa em PVS0.
- pvs0_lang: contém formalizações relacionadas com a maneira de expressar uma função em forma de 4-upla, onde o três primeiros elementos são interpretações de uma função PVS0 e o último é uma expressão PVS0.
- pvs0_prelude: especifica os naturais como um espaço de medidas e a relação menor que como uma ordem bem fundada.
- counting_pvs0_type: contém a formalização de que toda expressão PVS0 pode ser representada por um número natural.
- pvs0_pvs: contém a formalização que qualquer função pode ser expressa como uma expressão PVS0.

- pvs0_to_dg: contém a construção de grafo de contexto de chamado a partir de uma expressão PVS0.
- pvs0_bool: contém a representação de captura de expressões booleanas de PVS0.
- PVS0Expr: contém a especificação da gramática PVS0.

Os doze arquivos da biblioteca CCG são:

- scp_to_ccg: contém a formalização de que grafos de contexto de chamado implemntam o princípio da mudança de tamanho.
- scp_to_ccg_alt: contém a mesma formalização acima mas usando lemas diferentes
- ccg_to_mwg: contém o teorema de equivalênica entre CCG e MWG.
- bounding_circuits: contém um teorema de que circuitos limitados implica em terminação por MWG.
- ccg: contém lemas auxiliares relacionados a CCG.
- finite_sets_of_sets: contém propriedades relacionadas a conjuntos finitos de conjuntos.
- ccg_def: contém a definição de CCG.
- scp: contém a definição do princípio da mudança de tamanho.
- matrix_wdg: contém definição de MWG e outros critérios de terminação.
- cc_def: contém a definição de contexto de chamado.
- measures: contém definições e lemas relacionados com as medidas de MWG.

Ainda os arquivos pvs0_to_ccg, ccg_to_pvs0 e scp_iff_pvs0 importam ccg_def e scp.

Da biblioteca PVS0, os arquivos que podem ser reusados são measure_termination, measure_termination_defs e pvs0_props tanto as propriedades formalizadas quanto as definições funcionais.

Uma das características de PVS0 é que ele funciona apenas com um único tipo, tanto de entrada como de saída. É preciso simular o funcionamento de outros tipos para descrever como funções e operações cujos tipos de entrada e saída diferentes se comportam.

Existem diferentes abordagens para definir terminação dependendo do modelo computacional. Alguns dos modelos computacionais são máquinas de Turing, funções recursivas, programas funcionais e imperativos, e sistemas de reescrita de termos (*term rewriting system* ou TRS).

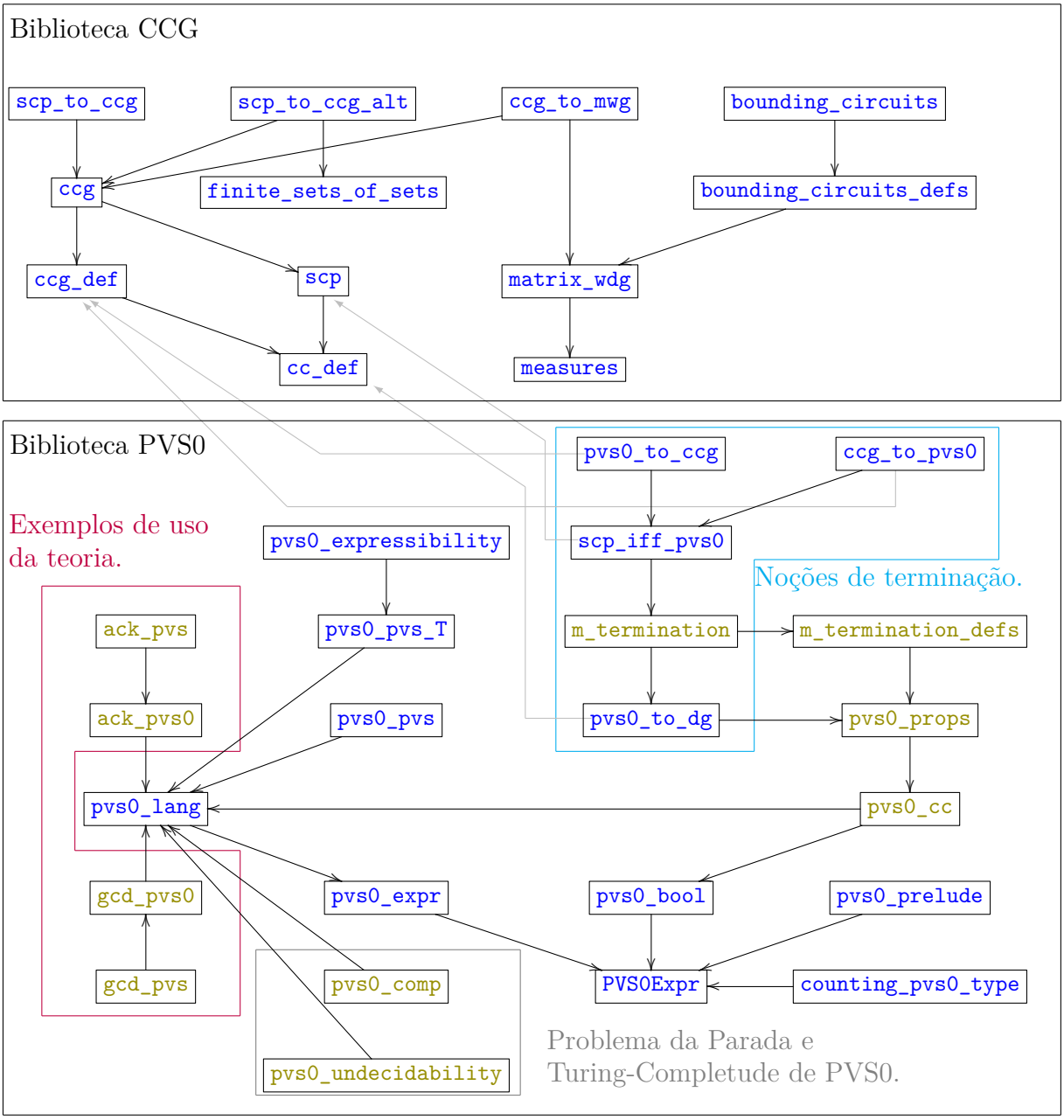


Figura 1.1: Bibliotecas PVS0 e CCG

Uma das maneiras de definir terminação é com o princípio da mudança de tamanho, aplicado em programação funcional [16] e em TRSs [19]. O princípio da mudança de tamanho diz que, em um programa, no qual os argumentos das chamadas recursivas decrescem em relação aos parâmetros das funções, cada sequência infinita de chamadas (seguindo o fluxo do controle do programa) deveria levar a uma sequência de valores de dados que decrescem infinitamente. Para programas funcionais de primeira ordem e com recursão mútua, [16] introduz a ideia de grafos de mudança de tamanho, em que cada argumento de uma função recursiva é representada como um vértice e cada vez em que ocorre uma chamada de outra ou da mesma função recursiva, mapeia-se em uma aresta como ocorreu a mudança verificando decrescimento ou igualdade por relações bem fundadas. Cada chamada recursiva produz um grafo diferente, e os grafos podem ser compostos, verificando assim o decrescimento geral de cada função.

O princípio da mudança de tamanho foi extendido para provar terminação por avaliação *mais interna* (*innermost termination*, que é quando a redução por uma regra de reescrita começa do subtermo mais interno a direita) em TRSs [19]. Esse princípio está implementado na biblioteca CCG em estruturas como grafos com matrizes de medidas de Avelar [4], e grafos de contextos de chamados de Manolios e Vroon [17], disponível nas bibliotecas de PVS da NASA [1].

A noção de contexto de chamado introduzida por Manolios e Vroon [17] é a mesma deste documento exceto pelo fato que eles consideram recursão mútua e que se define não terminação por chamadas infinitas, onde depois de chamadas infinitas retorna-se ao símbolo de não terminação $\perp$. Para isso, eles usam um operador de ponto fixo ξ, que simula a recursão mútua, e a ideia de limite tendendo ao infinito de aplicações de ξ para dar a possibilidade das definições de função ser uma definição de função parcial, que é como Vroon e Manolios chamam as funções não-terminantes. Cada chamado de contexto extraído de um código fonte é transformado em um vértice e cada possibilidade de chamado é uma aresta. Cada aresta é etiquetada de acordo com o decrescimento existente entre os chamados por uma ordem bem fundada e para um espaço de medidas de acordo com três possibilidades: decresce, decresce ou permanece igual e não se pode determinar se ou não decresce. Depois, verifica-se se em todas as possibilidades de circuito há um decrescimento.

Já MWG de Avelar [4] é uma abstração de CCG, a diferença é que no lugar de etiquetar as arestas com uma possibilidade de decrescimento, etiquetam-se com uma matriz, chamada de matriz de medidas. Nessa matriz, há valores de medida, 1, 0, -1, que respectivamente representam as possibilidades em uma ordem bem fundada de decrescemento, decrescemento ou igualdade, e indefinido ou não decrescimento. Essa matriz é organizada de tal modo que um elemento na linha i e coluna j representa uma comparação entre

a função de medida de número i com a função de medida de número j aplicadas aos parâmetros da função e aos argumentos do chamado recursivo.

Avelar [4] mostra e formaliza que as noções de terminação usando MWG e CCG são equivalentes entre si. Com as mesmas ideias de prova, as bibliotecas CCG está formalizada e disponível em [1].

Outra técnica de verificação de terminação em TRS é a de pares dependentes, implementado em [12]. Pares dependentes são tuplas advindas de regras de reescrita onde se aplicam conceitos semelhantes ao CCG.

1.1 Trabalhos relacionados

Em [3] os autores apresentam um técnica para verificação de terminação de sistemas de reescrita de termos (TRS) baseada em dígrafos com uma quase-ordem bem fundada, usando a noção de *dependency-pairs*, que são pares de termos conectados por uma regra de redução do TRS. Esses pares de termos tem como símbolo raiz um símbolo não constructor, estando o primeiro no lado esquerdo e o seguno no lado direito de cada regra. Em [13] implementa-se a técnica descrita em [3], gerando um ambiente denominado AProVE de verificação de terminação de especificações em TRS, programas imperativos, lógicos e funcionais. Em [10] e [11] apresenta-se a ferramenta KITTel que verifica terminação de programas em C fazendo uma tradução desses programas para TRSs. Em [6] é apresentado o *Size-change/MCNP*, que é uma implementação de provas de terminação via *monotonicity constraints*; esta é uma generalização do princípio da mudança de tamanho. Em [14] é apresentada uma ferramenta chamada TTT2 que analiza automaticamente terminação de sistemas de reescrita de termos de primeira ordem que também é baseada na noção de *dependency-pairs*.

Existem ainda bibliotecas de propriedades de terminação que auxiliam ferramentas como:

- a biblioteca CoLoR [5], implementada no provador de teorema Coq, que contém uma série de definições e teoremas relacionados com TRS e é usada pela ferramenta Rainbow e também pela AProVE, TTT2 e MatchBox;
- a biblioteca IsaFoR [2], implementada no provador de teorema Isabelle/HOL, contém várias técnicas de terminação por TRS e é usada pela ferramenta CeTA [20]
- Em Isabelle/HOL, foi formalizado o princípio da mudança de tamanho [15] porém não é usado por nenhuma ferramenta.
- Na ferramenta ACL2 o princípio da mudança de tamanho foi implementado [17] através de CCG mas não foi formalizado.

1.2 Motivação

Como terminação é importate para correção total de programas e é um problema indecídivel, é importante ter formalizadas as definições de terminação para ter total segurança de que as técnicas funcionam, pois uma demonstração em assistente de prova é mais forte que demonstrações em papel e lápis. Uma vez que se tem elas prontas é possível formular código fonte. Com isso se garante a correção de algoritmos baseados nessas técnicas. Também é importante caracterizar novas formas de se definir terminação, pois cada definição gera uma nova técnica de verificação para um programa semi-decidível (um que verifica uma reposta de "sim" ou "não sei") que pode ser comparada com outros em experimentos, como feito em [9].

1.3 Contribuições

A contribuição desse trabalho inclui:

- formalização de técnicas para se checar terminação por *ranking function*. Especificamente, formulação de métricas e funções auxiliares para formalização de equivalência entre *ranking function* e terminação semântica.
- Formalização do fato de que o modelo computacional usado, a linguagem PVS0, é Turing-Completo.
- Formalização do teorema de indecidibilidade do Problema da Parada para a linguagem PVS0.

1.4 Estrutura da dissertação

O Capítulo 2 apresenta provas clássicas do teorema de indecidibilidade do Problema da Parada para Máquinas de Turing e para a linguagem de máquinas com registros. O objetivo é apresentar um contexto padrão do tratamento deste problema no qual são utilizados elementos também necessários no tratamento da formalização de indecidibilidade para a linguagen PVS0, como são a necessidade de godelização dos programas PVS0, de forma que encodificações de programas PVS0 possam ser considerados parâmetros de entrada de programas PVS0 de forma univoca, e a argumentação por diagonalização que gera contradição a partir da suposição de existência de um programa PVS0 que resolve a questão de se um programa PVS0 para ou não.

O Capítulo 3 apresenta as especificações em PVS das noções de terminação semântica, conforme a semântica operacional da linguagem PVS0, assim como o critério de termina-

ção para PVS0 utilizado pela linguagem PVS, que é o de *ranking function*, implementado em PVS via geração automática de, assim denominados TCCs (type correctness conditions) de terminação. Essas condições essencialmente são conjecturas a serem demonstradas que garantem o decrescimento dos parâmetros em cada chamado recursivo.

O Capítulo 4 apresenta as formalizações de equivalência entre as noções de terminação apresentadas no capítulo anterior.

O Capítulo 5 apresenta a formalização da indecibilidade do Problema da Parada usando godelização, diagonalização e a equivalência entre terminação semântica e *ranking function* e também apresenta a formalização de Turing-Completude de PVS0 usando godelização. Mostram-se como a teoria pode contribuir para formalização da parada de funções especificadas em PVS tendo como exemplo a função de máximo divisor comum e a função de Ackermann.

O Capítulo 6 apresenta a conclusão do trabalho e os possíveis trabalhos futuros como a extensão de PVS0 para PVS1, que é uma linguagem que lidaria com diferentes tipos para entrada e saída.

Capítulo 2

O Problema da Parada

Neste capítulo, compara-se formulações clássicas do Problema da Parada para dois paradigmas diferentes: as Máquinas de Turing e as Máquinas com Registros. Nisso mostra-se que não importa o modelo de computação: a demonstração é praticamente a mesma, que é por diagonalização. Essa técnica serviu de base para para formalização da indecidibiildade do Problema da Parada para a linguagem PVS0 mostrado no Capítulo 5.

2.1 O Problema da Parada para máquinas de Turing

O Problema da Parada, também conhecido como *Halting Problem*, questiona se, dados uma descrição de Máquina de Turing e uma possível entrada para ela, a máquina para. Uma Máquina de Turing é uma máquina de estados, onde há uma fita infinita para a direita dividida em células. A máquina tem um cabeçote que lê um símbolo da fita e, dependendo do estado, escreve um símbolo, move uma célula para direita ou esquerda e muda de estado. Ela possui um estado inicial e, no começo, o cabeçote está no inicio da fita e há uma palavra nela. Máquinas de Turing podem transformar uma entrada em uma saída ou respoder se uma palavra pertence ou não a uma linguagem. Em termos formais seria a 6-upla $\langle Q, q_i, q, \Gamma, \Sigma, \delta \rangle$ onde:

- Q é um conjunto finito não vazio de estados;
- $q_i \in Q$ é o estado inicial;
- $q \in Q$ é o estado de aceitação;
- Γ é o alfabeto de entrada da máquina e possui o símbolo de branco $\sqcup$;
- Σ é o alfabeto de escrita;
- $\delta : (Q \times (\Gamma \cup \Sigma)) \rightarrow (Q \times (\Gamma \cup \Sigma) \times \{>, <\})$ é a função de transição, o símbolo $>$ indica movimento para direita, o símbolo $<$ indica movimento para esquerda. A

função δ é uma função parcial. O estado resultante é o próximo estado e o símbolo resultante é o símbolo a ser escrito antes de mudar de estado.

A representação de uma computação é dada por uma sequência de configurações. Assim, uma configuração é dada por uma sequência na forma $s \in (\Gamma \cup \Sigma)^* Q (\Gamma \cup \Sigma)^+$. Cada configuração representa o estado em que a máquina está, o que está escrito na fita e onde está o cabeçote. Uma configuração inicial pertence a $q_i \Gamma^+$. Em uma configuração, o estado fica ao lado esquerdo do símbolo onde o cabeçote da fita está localizado e as sequências que estão a esquerda e a direita do estado, quando concatenados nessa ordem, formam o que está escrito na fita. Quando o estado está totalmente a esquerda e a função de transição resulta em um movimento para esquerda, o cabeçote muda de estado, escreve o novo símbolo na fita, mas o cabeçote mantém-se no mesmo lugar. Quando o estado está mais à direita (falta um símbolo para ficar totalmente à direita da configuração) e a função de transição pede para ir para à direita, é adicionado um símbolo de branco a direita da configuração e a transição é feita normalmente. E quando o estado está mais à direita, o símbolo a direita do estado é branco e a função de transição pede para escrever branco e ir à esquerda, então a próxima configuração será com esse último símbolo de branco apagado e o próximo estado é colocado à esquerda. Caso o contrário a tudo isso, a mudança de configuração será para todo $w_1 \in (\Gamma \cup \Sigma)^*$, $w_2 \in (\Gamma \cup \Sigma)^+$, $a, b, c \in (\Gamma \cup \Sigma)$ e $q_1, q_2 \in Q$:

de $w_1\ q_1\ a\ w_2$ para $w_1\ b\ q_2\ w_2$ se $\delta(q_1, a) = (q_2, b, >)$ e

de $w_1\ c\ q_1\ a\ w_2$ para $w_1\ q_2\ c\ b\ w_2$ se $\delta(q_1, a) = (q_2, b, <)$

Quando $\delta(q_1, a)$ atinge um estado de aceitação, a máquina aceita a palavra inicial. Mas quando $\delta(q_1, a)$ não está definida, a máquina rejeita a palavra inicial. Ainda pode acontecer da máquina continuar a processar e não parar. Sempre que uma máquina M aceita uma palavra i pertencente a uma linguagem, L e caso contrário rejeita ou não para, é dito que M reconhece L. Mas caso a máquina M sempre pare, é dito que M decide L. Se não há máquina que decide L, L é chamada indecidível.

Como exemplo de linguagem decidível tem-se a linguagem das palavras duplicadas para o alfabeto 0,1. Assim a 6-upla:

- $Q := \{1, 2, 3, 4, 5, 6, 7, 8\}$
- 1 como estado inicial
- 8 como estado de aceitação
- $\Gamma := \{0, 1, \sqcup\}$
- $\Sigma := \{X\}$

- $\delta(1,1) = (2, X, >)$
  - $\delta(1,0) = (6, X, >)$
  - $\delta(2,1) = (2, 1, >)$
  - $\delta(2,0) = (2, 0, >)$
  - $\delta(2, \sqcup) = (3, \sqcup, <)$
  - $\delta(2, X) = (3, X, <)$
  - $\delta(3,1) = (4, X, <)$
  - $\delta(4, X) = (8, X, >)$
  - $\delta(4,1) = (5, 1, <)$
  - $\delta(4,0) = (5, 0, <)$
  - $\delta(5,1) = (5, 1, <)$
  - $\delta(5, X) = (1, X, >)$
  - $\delta(6,1) = (6, 1, >)$
  - $\delta(6,0) = (6, 0, >)$
  - $\delta(6, \sqcup) = (7, \sqcup, <)$
  - $\delta(6, X) = (7, X, <)$
  - $\delta(7,0) = (4, X, <)$

Um exemplo de execução para a entrada 101101 com o estado atual com uma chave acima dele.

- $\overbrace{1}^{\frown} 101101$
- $X \overbrace{2}^{\frown} 01101$
- $X0 \overbrace{2}^{\frown} 1101$
- $X01 \overbrace{2}^{\frown} 101$
- $X011 \overbrace{2}^{\frown} 01$
- $X0110 \overbrace{2}^{\frown} 1$
- $X01101 \overbrace{2}^{\frown} \sqcup$
- $X0110 \overbrace{3}^{\frown} 1$
- $X011 \overbrace{4}^{\frown} 0X$
- $X01 \overbrace{5}^{\frown} 10X$
- $X0 \overbrace{5}^{\frown} 110X$
- $X \overbrace{5}^{\frown} 0110X$
- $\overbrace{5}^{\frown} X0110X$
- $X \overbrace{1}^{\frown} 0110X$
- $XX \overbrace{6}^{\frown} 110X$
- $XX1 \overbrace{6}^{\frown} 10X$
- $XX11 \overbrace{6}^{\frown} 0X$
- $XX110 \overbrace{6}^{\frown} X$
- $XX11 \overbrace{7}^{\frown} 0X$
- $XX1 \overbrace{4}^{\frown} 1XX$
- $XX \overbrace{5}^{\frown} 11XX$
- $X \overbrace{5}^{\frown} X11XX$
- $XX \overbrace{1}^{\frown} X1XX$
- $XXX \overbrace{2}^{\frown} 1XX$
- $XXX1 \overbrace{2}^{\frown} XX$
- $XXX \overbrace{3}^{\frown} 1XX$
- $XX \overbrace{4}^{\frown} XXXX$
- $XXX \overbrace{8}^{\frown} XXX$

Aqui, define-se $\langle M \rangle$ como sendo a descrição da máquina M em termos de uma palavra em Γ^*. Essa descrição é chamada de godelização.

Logo, em termos de linguagem, o problema da parada seria:

$HALT = \{\langle \langle M \rangle, i \rangle |\ M \text{ para com a entrada } i\}$

Como é conhecido, Máquinas de Turing são modelos matemáticos que descrevem o funcionamento de programas. Portanto, se existe uma Máquina de Turing que decide a linguagem acima, há como se construir um compilador que verifique se programas terminam. Suponha que essa máquina seja M_{HALT}. Com ela pode-se contruir a máquina M_0 com a seguinte descrição:

Com a entrada s:

- Se M_{HALT} aceita s, M_0 entra em ciclo infinito.
- Caso contrário, M_0 aceita s.

Ao passar a própria descrição de M_0 como entrada de M_0:

- Se M_{HALT} aceita $\langle M_0 \rangle$, então M_0 não para. Logo M_{HALT} rejeita $\langle M_0 \rangle$.
- Se M_{HALT} rejeita $\langle M_0 \rangle$, então M_0 para. Logo M_{HALT} aceita $\langle M_0 \rangle$.

Portanto, uma máquina que decide a linguagem $HALT$ não existe. Assim, não há como construir um compilador que determine se um programa termina para alguma entrada, pois máquina de Turing é um modelo equivalente a programas e se existe um compilador com esse requisito, então há uma máquina de Turing correspondente. Nota-se que precisa existir uma correpondência entre uma palavra em Γ^* e uma Máquina de Turing qualquer. Se a fita for finita, essa correspondência não existe, pois o número de Máquinas de Turing são infinitas. Nesse caso, o número de configurações é finito e portanto, é contável. Digamos que a máquina tenha um tamanho de fita igual à n. O número total de configurações será $n \cdot |Q| \cdot |\Gamma \cup \Sigma|^n$. Quando se ultrapassa esse valor, significa que alguma configuração repetiu e que a sequência de configurações entre as repetições também irá se repetir. Nesse caso, basta contar o número de configurações para detectar se a máquina para ou não.

2.2 O Problema da Parada para programas com registros

Outro formalismo para definir o Problema da Parada são programas com registros. Segundo a tese de Church-Turing toda função computável pode ser computada por uma máquina de Turing. Nesse caso programas com registros são equivalentes com máquinas de Turing. Para definir programas com registros é preciso dizer que ele trabalha com o alfabeto $\mathbb{A} = \{a_0, \cdots, a_s\}$. Esse tipo de programa é executado em uma máquina com registros $R_0, \cdots, R_h$, em que cada registro contém uma palavra de $\mathbb{A}^*$. O símbolo $\square$ representa a palavra vazia. Um programa sobre $\mathbb{A}$ consiste em uma senquência finita das seguintes instruções rotuladas com um natural n:

1. Intrução de adição: n `LET` $R_i = R_i + a_j$

   Adiciona a letra a_j ao final do registro R_i.

2. Instrução de subtração: n `LET` $R_i = R_i - a_j$

   Retira a letra a_j do final da palavra R_i caso ela esteja lá. Caso contrário, não faz nada.

3. Instrução de salto n `IF` $R_i = \square$ `THEN` m' `ELSE` m_0 `OR` $\cdots$ `OR` m_s

   Se o registro R_i for a palavra vazia vá para a instrução m'. Caso contrário, se a palavra termina com a_0 (respectivamente com $a_1, \cdots, a_s$) vá para m_0 (respectivamente para $m_1, \cdots, m_s$)

4. Instrução de impressão: n `IMPRIMA`

   Imprima o que está no registro R_0.

5. Instrução de parada: n `PARA`

   Pare de executar.

Um programa P sobre $\mathbb{A}$ inicia sua computação com o registro R_0 com uma palavra $\eta \in \mathbb{A}^*$ e os outros registros com a palavra vazia. Caso P inicie com uma palavra η e atinga uma instrução de parada, escreve-se $P : \eta \to$ `PARA`. Caso contrário, $P : \eta \to \infty$. O programa P é dito decidível se $\forall \eta \; P : \eta \to$ `PARA` e se decide se uma palavra pertence a uma linguagem $\mathbb{L} \subseteq \mathbb{A}^*$, respondendo a palavra vazia no registro R_0 caso a entrada pertença e uma palavra não vazia caso contrário. Supondo que existe uma godelização das palavras de $\mathbb{A}$, ou seja, para cada programa P existe uma forma de representá-lo η_P que é uma palavra de $\mathbb{A}$, o Problema da Parada se caracteriza em decidir o conjunto abaixo:

$$\Pi_{\texttt{PARA}} = \{\eta_P | P : \eta_P \to \texttt{PARA}\}$$

Suponha que P decide $\Pi_{\texttt{PARA}}$. Construa P_1 onde caso P decide que a entrada para então crie uma instrução que não pare. Caso contrário, então pare. Dessa forma:

$$P_1 : \eta_P \to \infty \text{ se } P : \eta_P \to \texttt{PARA}$$

$$P_1 : \eta_P \to \texttt{PARA} \text{ se } P : \eta_P \to \infty$$

Se η_{P_1} for executado em P_1 então:

$$P_1 : \eta_{P_1} \to \infty \text{ se } P_1 : \eta_{P_1} \to \texttt{PARA}$$

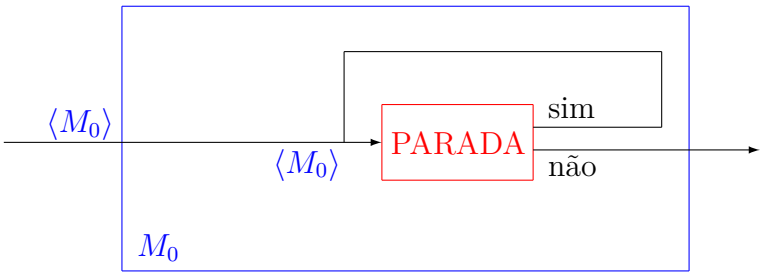


Figura 2.1: Esquema de prova de indecidibilidade do Problema da Parada

$$P_1 : \eta_{P_1} \rightarrow \texttt{PARA} \text{ se } P_1 : \eta_{P_1} \rightarrow \infty$$

Que é contradição. Assim o problema da parada é indecidível para programas com registros. Essa é uma das características de tudo o que é Turing-Completo.

2.3 O Problema da Parada em Geral

A técnica de demonstração de indecidibilidade do Problema da Parada aplicada neste capítulo para os modelos computacionais de Máquinas de Turing e Programas com Registros consiste dos seguintes passos:

- Mostrar que há uma godelização da linguagem.
- Supor que existe uma máquina/programa que verifica a parada de outras máquinas/programas.
- Usar a máquina/programa que verifica parada para criar outra máquina/programa em que se para entrada há parada, então não pare e caso contrário, pare.
- Executar a máquina/programa criada usando como entrada ela mesma para gerar contradição.

A formalização de indecidiblidade do Problema da Parada para a linguagem PVS0 segue os mesmos passos comforme a figura 2.1, onde $\langle M_0 \rangle$ é a descrição da máquina M_0.

Capítulo 3

Noções de Terminação

3.1 A linguagem PVS0

Conforme dito na introdução, a biblioteca PVS0 contém 269 propriedades formalizadas. Antes de se definir terminação, é preciso dizer qual é o modelo computacional. O modelo aqui é baseado no paradigma funcional. Esse modelo permite expressar programas através de funções. Pode ocorrer do processamento da função não terminar, pois como é de conhecimento comum, funções podem estar definidas em termos delas mesmas (a isso é chamado recursão) e na suas avaliações elas podem se subavaliar várias vezes sem terminar. Como o modelo funcional é equivalente a máquinas de Turing, então também não se pode criar um função que verifica se outras estão definidas ou não.

No paradigma funcional foi definido um modelo funcional mínimo, que inclui justo os elementos sintáticos e semânticos que uma linguagem precisa para ser Turing Completa (consegue-se codificar máquinas de Turing). É dita que ela é mínima porque contém o minimo de elementos para ser Turing Completa. Por ser minima, ela permite simplificação das formalizações, com menos casos a serem analisados.

A gramática para a linguagem PVS0 está dada pelas seguintes regras BNF:

$$
\begin{aligned}
\mathcal{E}xpr \quad ::= \quad & rec(\mathcal{E}xpr) \\
& |op1(\mathcal{E}xpr) \\
& |op2(\mathcal{E}xpr, \mathcal{E}xpr) \\
& |ite(\mathcal{E}xpr, \mathcal{E}xpr, \mathcal{E}xpr) \\
& |vr \\
& |cnst
\end{aligned}
$$

Acima, rec é a recursão, $op1$ é um dos símbolos de operador unário, $op2$ é um dos símbolos de operador binário, vr é o símbolo de variável (a linguagem trabalha apenas

com uma única variável), *cnst* é um dos símbolos de constante e *ite* é o símbolo de ramificação `IF THEN ELSE`.

Em PVS, a gramática da linguagem funcional foi especificada como uma estrutura abstrata de dados:

```
PVSOExpr[T:TYPE+] :  DATATYPE
BEGIN
   cnst(get_val:T) : cnst?  :  PVSOExpr
   vr :  vr?  :  PVSOExpr
   op1(get_op:nat,get_arg:PVSOExpr) :  op1?  :  PVSOExpr
   op2(get_op:nat,get_arg1,get_arg2:PVSOExpr) :  op2?  :  PVSOExpr
   rec(get_arg:PVSOExpr) :  rec?  :  PVSOExpr % recursive call
   ite(get_cond,get_if,get_else:PVSOExpr) :  ite?  :  PVSOExpr
END PVSOExpr
```

Acima, `T` é o tipo dos elementos que serão avaliados pela linguagem, tanto como entrada quanto como saída e ele deve ser não vazio, como está escrito em `T:TYPE+` logo na primeira linha. Ainda acima, nota-se que `op1` e `op2` no primeiro argumento recebe um número natural. Esse natural representa um símbolo da gramática e é mapeado em uma função para a interpretação dos operadores unários e binários da linguagem como descrito abaixo por $\mathfrak{I}$. Da mesma maneira, `cnst` contém um argumento que é interpretado como uma constante.

3.2 Noções de terminação semântica

Tem-se os conjuntos não vazios de símbolos de constantes $CONST$, de variáveis $\mathcal{V}ar$, de operadores binários $OP2$ e de operadores unários $OP1$, todos disjuntos. Para a interpretação, usa-se um conjunto não vazio de valores $\mathcal{V}al$, que são valores tanto de entrada quanto de saída. Assim, a interpretação $\mathfrak{I} : (CONST \to \mathcal{V}al) \cup (OP1 \to \mathcal{V}al \to \mathcal{V}al) \cup (OP2 \to (\mathcal{V}al \times \mathcal{V}al) \to \mathcal{V}al)$, funciona da seguinte forma (representando $\mathfrak{I}(m)$ como $m^{\mathfrak{I}}$):

- Se $c \in CONST$, então $c^{\mathfrak{I}} \in \mathcal{V}al$ é um valor correpondente a constante c.
- Se $g \in OP1$, então $g^{\mathfrak{I}} \in \mathcal{V}al \to \mathcal{V}al$ é a função unária correspondente a g.
- Se $h \in OP2$, então $h^{\mathfrak{I}} \in (\mathcal{V}al \times \mathcal{V}al) \to \mathcal{V}al$ é a função binária correspondente a h

Para avaliar uma função, é preciso avaliar os argumentos. Assim, define-se também a relação semântica $\varepsilon : \mathcal{E}xpr \times \mathcal{E}xpr \times \mathcal{V}al \times \mathcal{V}al \to$ `bool` onde e é a expressão avaliada, e_f é a expressão que corresponde à recursão, β é a entrada da avaliação e ν é a saída da

avaliação como sendo:

$$
\begin{aligned}
\varepsilon(e, e_f, \beta, \nu) \;:=\; & \textbf{CASES } e \textbf{ OF} \\
& cnst: \;\; \nu = cnst^{\mathfrak{I}}; \\
& vr: \;\; \nu = \beta; \\
& op1(e_1): \;\; \exists \nu_1 \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \\
& \qquad \nu = op1^{\mathfrak{I}}(\nu_1); \\
& op2(e_1, e_2): \;\; \exists \nu_1, \nu_2 \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \varepsilon(e_2, e_f, \beta, \nu_2) \wedge \\
& \qquad \nu = op2^{\mathfrak{I}}(\nu_1, \nu_2); \\
& ite(e_1, e_2, e_3): \;\; \exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \\
& \qquad \textbf{IF } \nu_1 \textbf{ THEN } \varepsilon(e_2, e_f, \beta, \nu) \textbf{ ELSE } \varepsilon(e_3, e_f, \beta, \nu); \\
& rec(e_1): \;\; \exists \beta' \in \mathcal{V}al : \varepsilon(e_1, e_f, \beta, \beta') \wedge \\
& \qquad \varepsilon(e_f, e_f, \beta', \nu)
\end{aligned}
$$

O pseudo-código acima avalia através de *lazy evalutation* (do inglês avaliação preguiçosa), técnica que consiste em avaliar apenas quando necessário. Por exemplo, se há duas funções $f : \mathbb{N} \to Bool$ e $g : \mathbb{N} \to Bool$, é avaliado $f(5) \wedge g(6)$, onde $f(5)$ é avaliado como falso e $g(6)$ é avaliado como verdadeiro, mas $f(5)$ é avaliado primeiro tornando a expressão $f(5) \wedge g(6)$ falsa.

O predicado ε descreve como é avaliada uma expressão e, que pode conter a função recursiva dada pela expressão e_f, cujo argumento é dado pelo valor β e tem como resultado ν. Mas se a recursão não para, a avaliação também não, e o resultado não existirá. Como exemplo de uso do predicado tem-se a função de máximo divisor comum chamado aqui de *gcd* que trabalha no tipo $\mathbb{N} \times \mathbb{N}$:

$$gcd := ite(Z_{ero}(vr), S_{um}(vr), ite(G_{eq}(vr), rec(S_{ub}(vr)), rec(P_{er}(vr))))$$

As interpretações dos símbolos acima são:

- $Z^{\mathfrak{I}}_{ero}(\langle m, n\rangle) := m = 0 \vee n = 0$
- $S^{\mathfrak{I}}_{um}(\langle m, n\rangle) := (m + n, 0)$
- $G^{\mathfrak{I}}_{eq}(\langle m, n\rangle) := n \geq m$
- $S^{\mathfrak{I}}_{ub}(\langle m, n\rangle) := (m, n - m)$
- $P^{\mathfrak{I}}_{er}(\langle m, n\rangle) := (n, m)$

Ou seja, *gcd* modela a função 1

Ao avaliar *gcd* com o argumento $(36, 9)$, espera-se como resposta $(9, 0)$:

$$\varepsilon(gcd, gcd, (36, 9), (9, 0))$$

Algoritmo 1: gcd

Input: $(m : \mathbb{N}, n : \mathbb{N})$
Output: $(a, 0)$ onde a é máximo divisor comum entre m e n se ambos não são nulos e 0 caso contrário

```
1 if m = 0 ∨ n = 0 then
2     (m + n, 0);
3 else if n ≥ m then
4     gcd(m, n − m);
5 else
6     gcd(n, m);
```

Outro exemplo para de uso do predicado é a função *dobro* que retorna o dobro de um número natural:

$$dobro := ite(Z_{ero}(vr), Z_0, S_{um2}(rec(M_{inus1}(vr))))$$

As interpretações dos símbolos são:

- $Z^{\mathfrak{I}}_{ero}(m) := m = 0$
- $Z^{\mathfrak{I}}_0 := 0$
- $S^{\mathfrak{I}}_{um2}(m) := m + 2$
- $M^{\mathfrak{I}}_{inus1}(m) := m - 1$

Esse código acima modela a função 2.

Algoritmo 2: dobro

Input: $m : \mathbb{N}$
Output: $2 \times m$

```
1 if m = 0 then
2     0;
3 else
4     2 + dobro(n − 1);
```

Ao avaliar *dobro* com argumento 3 espera-se 6 como resposta:

$$\varepsilon(dobro, dobro, 3, 6)$$

Outro exemplo seria a função $fibonacci$:

$$fibonacci := ite(Z_{ero}(vr), 0_S, ite(Z_{ero}(op1_0(vr)), 1_S, op2_0(rec(op1_0(vr)), rec(op1_1(vr)))))$$

As interpretações dos símbolos são:

- $Z_{ero}^{\mathfrak{I}}(m) := m = 0$
- $0_S^{\mathfrak{I}} := 0$
- $1_S^{\mathfrak{I}} := 1$
- $op1_0^{\mathfrak{I}}(m) := m - 1$
- $op1_1^{\mathfrak{I}}(m) := m - 2$
- $op2_0^{\mathfrak{I}}(m, n) := m + n$

O código acima modela a função 3.

Algoritmo 3: fibonacci

Input: $m : \mathbb{N}$
Output: o m-ésimo número de fibonacci

```
1 if m = 0 then
2     0;
3 else if m − 1 = 0 then
4     1;
5 else
6     fibonacci(m − 1) + fibonacci(m − 2);
```

Assim, avaliar $fibonacci$ de 5 é 5:

$$\varepsilon(fibonacci, fibonacci, 5, 5)$$

Em PVS, esse predicado foi representado como uma definição indutiva em PVS.

```
semantic_rel_expr(false_val,eval_op1,eval_op2)
(expr,body:PVSOExpr,env:Val,val:Val):  INDUCTIVE bool =
(cnst?(expr) AND val = get_val(expr)) OR

(vr?(expr) AND val = env) OR

(op1?(expr) AND EXISTS (valarg:Val) :
semantic_rel_expr(false_val,eval_op1,eval_op2)
```

```
(get_arg(expr),body,env,valarg) AND
val = eval_op1(get_op(expr))(valarg)) OR

(op2?(expr) AND EXISTS (valarg1,valarg2:Val) :
semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_arg1(expr),body,env,valarg1) AND
semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_arg2(expr),body,env,valarg2) AND
val = eval_op2(get_op(expr))(valarg1,valarg2)) OR

(ite?(expr) AND EXISTS (valarg:Val) :
semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_cond(expr),body,env,valarg) AND IF false_val \= valarg THEN
semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_if(expr),body,env,val)
ELSE semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_else(expr),body,env,val) ENDIF) OR

(rec?(expr) AND EXISTS (valarg:Val) :
semantic_rel_expr(false_val,eval_op1,eval_op2)
(get_arg(expr),body,env,valarg) AND
semantic_rel_expr(false_val,eval_op1,eval_op2)
(body,body,valarg,val))
```

Assim, uma função representada pela expressão e_f é terminante se obedece ao predicado de terminação semântica T_ε abaixo, ou seja, quando para qualquer entrada a função produz um resultado.

$T_\varepsilon : \mathcal{E}xp \to \texttt{bool}$, definida por:

$$T_\varepsilon(e_f) := \forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon(e_f, e_f, \beta, \nu)$$

Em PVS tem-se o seguinte:

```
terminates_expr(false_val,eval_op1,eval_op2)(expr,body:Expr) :  bool =
FORALL (env:Val):
EXISTS (val:Val) :
semantic_rel_expr(false_val,eval_op1,eval_op2)(expr,body,env,val)
```

Pode-se, também, medir a quantidade de níveis de chamadas recursivas na execução. Se, para todas as entradas, o comprimento de cadeias de chamadas recursivas for finito,

então a função para. Para isso tem-se a definição da funcão $\chi : \mathbb{N} \times \mathcal{E}xp \times \mathcal{E}xp \times \mathcal{V}al \to \mathcal{V}al \cup \{\diamondsuit\}$ onde $\diamondsuit \notin \mathcal{V}al$.

$$
\begin{array}{rcl}
\chi(i, e, e_f, \beta) & := & \textbf{IF } i = 0 \textbf{ THEN } \diamondsuit \\
& & \textbf{ELSE } \textbf{ CASES } e \textbf{ OF} \\
& & \quad cnst: \; cnst^{\mathfrak{I}}; \\
& & \quad vr: \; \beta; \\
& & \quad op1(e_1): \; \texttt{IF } \chi(i, e_1, e_f, \beta) \neq \diamondsuit \\
& & \qquad \textbf{THEN } op1^{\mathfrak{I}}(\chi(i, e_1, e_f, \beta)) \\
& & \qquad \textbf{ELSE } \diamondsuit; \\
& & \quad op2(e_1, e_2): \; \textbf{IF } \chi(i, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(i, e_2, e_f, \beta) \neq \diamondsuit \\
& & \qquad \textbf{THEN } op2^{\mathfrak{I}}(\chi(i, e_1, e_f, \beta), \chi(i, e_2, e_f, \beta)) \\
& & \qquad \textbf{ELSE } \diamondsuit; \\
& & \quad ite(e_1, e_2, e_3): \; \textbf{IF } \chi(i, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
& & \qquad \textbf{IF } \chi(i, e_1, e_f, \beta) \textbf{ THEN } \chi(i, e_2, e_f, \beta) \\
& & \qquad \textbf{ELSE } \chi(i, e_3, e_f, \beta); \\
& & \qquad \textbf{ELSE } \diamondsuit; \\
& & \quad rec(e_1): \; \textbf{IF } \chi(i, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
& & \qquad \chi(i-1, e_f, e_f, \beta'), \\
& & \qquad \text{onde } \beta' = \chi(i, e_1, e_f, \beta) \\
& & \qquad \textbf{ELSE } \diamondsuit
\end{array}
$$

Na função $\chi(i, e, e_f, \beta)$, i é o número máximo da altura da árvore de chamadas recursivas, e é a expressão a ser avaliada, e_f é a expressão que representa a função recursiva e β é o argumento. O valor de i é a altura máxima da árvore de recursão. Nota-se que no caso recursivo é o único local em que i é decrementado, pois ele conta o número de níveis recursivos que pode ser alcançado. Quando o número de chamadas da recursão está ultrapassando esse limite, a função χ retorna $\diamondsuit$. Em PVS ela foi representada como a função recursiva abaixo.

```
eval_expr(false_val,eval_op1,eval_op2)
(i:nat,expr,body:Expr,env:Val) :  RECURSIVE Maybe[Val]
=
IF i = 0 Then None
ELSE CASES expr OF
     cnst(v):  Some(v),
     vr:  Some(env),
     op1(op,arg):
```

```
      LET v = eval_expr(false_val,eval_op1,eval_op2)(i,arg,body,env)
      IN
      IF some?(v) THEN Some(eval_op1(op)(val(v))) ELSE None ENDIF,
    op2(op,arg1,arg2):
      LET v1 = eval_expr(false_val,eval_op1,eval_op2)(i,arg1,body,env),
          v2 = eval_expr(false_val,eval_op1,eval_op2)(i,arg2,body,env)
      IN
      IF some?(v1) AND some?(v2) THEN
        Some(eval_op2(op)(val(v1),val(v2)))
      ELSE None ENDIF,
    ite(cnd,arg1,arg2):
      LET vc = eval_expr(false_val,eval_op1,eval_op2)(i,cnd,body,env)
      IN
      IF some?(vc) THEN IF false_val \= val(vc) THEN
           eval_expr(eval_bool,eval_op1,eval_op2)(i,arg1,body,env)
      ELSE
         eval_expr(eval_bool,eval_op1,eval_op2)(i,arg2,body,env)
      ENDIF ELSE None
      ENDIF,
    rec(arg):
      LET v = eval_expr(eval_bool,eval_op1,eval_op2)(i,arg,body,env)
      IN
      IF some?(v) THEN
        eval_expr(eval_bool,eval_op1,eval_op2)(i-1,body,body,val(v))
      ELSE None ENDIF
ENDCASES ENDIF
```

Acima, o tipo `Maybe[Val]` representa o tipo $\mathcal{V}al \cup \{\diamondsuit\}$, com o elemento `None` representando $\diamondsuit$ e os elementos de `Val` passando para `Maybe[Val]` pela função `Some`.

Assim a função representada pela expressão e_f termina se, para todas as entradas, a árvore de recursão tem altura limitada, de acordo com o predicado de terminação semântica T_χ.

$T_\chi : \mathcal{E}xp \to$ `bool`, definida por:

$$T_\chi(e_f) := \forall\ \beta,\ \exists\ m \in \mathbb{N} :\ \textbf{Seja}\ \nu := \chi(m, e_f, e_f, \beta)\ \textbf{em}\ \nu \neq \diamondsuit\ \wedge\ \varepsilon(e_f, e_f, \beta, \nu)$$

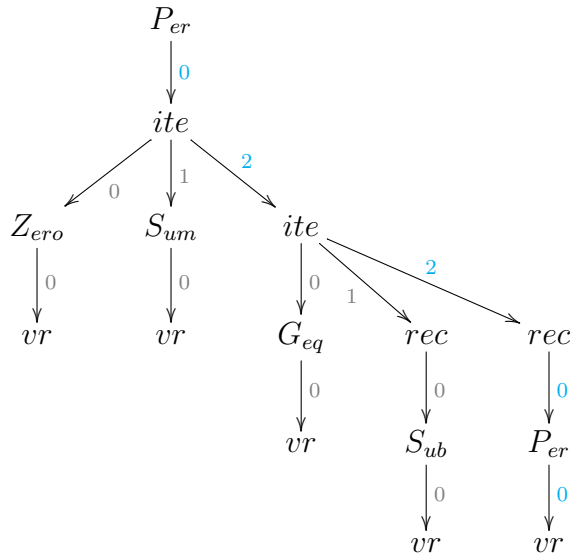
3.3 TCC Terminação

Outra abordagem para terminação é quando existe decréscimo em uma relação bem fundada entre os argumentos da função e os argumentos correspondentes da chamada recursiva, partindo da suposição que as condições para chegar até a chamada recursiva são verdadeiras. Essa idéia de terminação está implementada em diversos assistentes de prova, incluindo PVS e essa técnica se chama *ranking function* [22]. Logo, é preciso definir como extrair uma subexpressão por um caminho de execução em uma árvore de expressões, quais caminhos são válidos e armazenar as expressões booleanas até uma chamada recursiva. Para representar um caminho, uma lista com naturais é usada, lendo-a de trás para frente. Essa abordagem permite estender o caminho apenas acrescentando um inteiro válido a frente da lista, ao invés de percorrer toda lista e acrescentá-lo ao final. Abaixo, $[T]$ é o tipo lista de elementos de tipo T, $[]$ é a lista vazia, $[n : l]$ é uma lista cujo primeiro elemento é n e tem como cauda l, rac é a função que toma o último elemento de uma lista, rdc é a função que retira o último elemento e $++$ é a concatenação de listas.

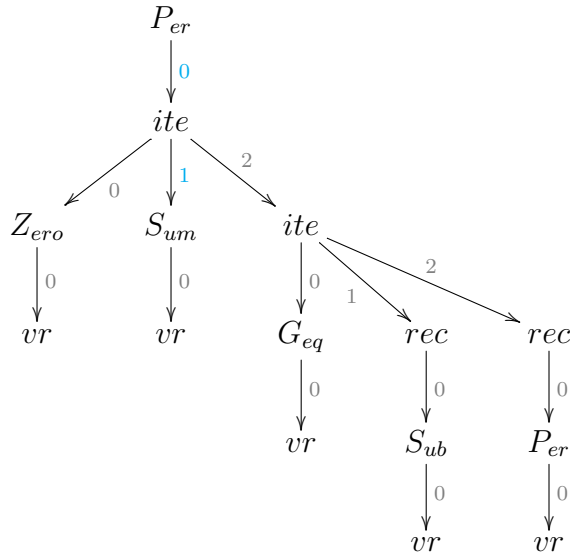
Assim, tem-se a definição de caminho válido, VP, em uma expressão. A função $VP(e)(l)$ verifica se a lista de naturais l é um caminho válido da expressão e. Logo, $VP : \mathcal{E}xpr \to [\mathbb{N}] \longrightarrow \texttt{bool}$ é definido como:

$$
\begin{array}{rrl}
VP(e)(l) & := & \textbf{CASES } e \textbf{ OF} \\
 & cnst: & l = []; \\
 & vr: & l = []; \\
 & op1(e_1): & l = [] \ \vee \\
 & & (rac(l) = 0 \wedge VP(e_1)(rdc(l))); \\
 & op2(e_1, e_2): & l = [] \ \vee \\
 & & (rac(l) = 0 \wedge VP(e_1)(rdc(l)))\vee \\
 & & (rac(l) = 1 \wedge VP(e_2)(rdc(l))); \\
 & ite(e_1, e_2, e_3): & l = [] \ \vee \\
 & & (rac(l) = 0 \wedge VP(e_1)(rdc(l)))\vee \\
 & & (rac(l) = 1 \wedge VP(e_2)(rdc(l)))\vee \\
 & & (rac(l) = 2 \wedge VP(e_3)(rdc(l))); \\
 & rec(e_1): & l = [] \ \vee \\
 & & (rac(l) = 0 \wedge VP(e_1)(rdc(l)))
\end{array}
$$

Usando a função VP define-se, para uma expressão e, o tipo $VP(e)$, que é o tipo de todas as listas de naturais l tais que $VP(e)(l)$. Uns exemplos de caminhos válidos na expressão $P_{er}(gcd)$ são $[1, 0]$, $[0, 1, 0]$ e $[0, 0, 2, 2, 0]$. Assim, valem $VP([1, 0])(P_{er}(gcd))$, $VP([0, 1, 0])(P_{er}(gcd))$ e $VP([0, 0, 2, 2, 0])(P_{er}(gcd))$. Abaixo, selecionou-se o caminho $[0, 0, 2, 2, 0]$ como um caminho válido em $P_{er}(gcd)$:



Da mesma forma, $[1, 0]$ é um caminho válido:



Caminhos válidos em uma expressão determinam posições em uma subexpressão.

Seja l uma lista de naturais e e uma expressão, tal que $VP(e)(l)$. Define-se a função que extrai subtermo $SA : \mathcal{E}xpr \times [\mathbb{N}] \to \mathcal{E}xpr$ como:

$$
\begin{array}{rcl}
SA(e,l) & := & \textbf{IF } l = [] \textbf{ THEN } e \\
& & \textbf{ELSE } \textbf{ CASES } e \textbf{ OF} \\
& & \quad op1(e_1): \; SA(e_1, rdc(l)); \\
& & \quad op2(e_1,e_2): \; \textbf{IF } rac(l) = 0 \textbf{ THEN } SA(e_1, rdc(l)) \\
& & \quad \textbf{ELSE } SA(e_2, rdc(l)); \\
& & \quad ite(e_1,e_2,e_3): \; \textbf{IF } rac(l) = 0 \textbf{ THEN } SA(e_1, rdc(l)) \\
& & \quad \textbf{ELSE IF } rac(a) = 1 \textbf{ THEN } SA(e_2, rdc(l)) \\
& & \quad \textbf{ELSE } SA(e_3, rdc(l)); \\
& & \quad rec(e_1): \; SA(e_1, rdc(l)); \\
& & \quad \textbf{ELSE}: \; e
\end{array}
$$

Como exemplo de extração de subtermos tem-se:

- $SA(P_{er}(gcd), [1,0]) = S_{um}(vr)$
- $SA(P_{er}(gcd), [0,1,0]) = vr$
- $SA(P_{er}(gcd), [2,2,0]) = rec(P_{er}(vr))$

Nisso é possível selecionar condições de acordo com um caminho de execução. Antes de se extrair essas condições de caminho, é preciso uma estrutura de dados que diga se aquele elemento foi retirado ao seguir por um `THEN` ou por um `ELSE` para que a avaliação da lista seja coerente. Nesse sentido, criou-se a estrutura de dados de tipo $\mathcal{E}xprbool$, com duas primitivas:

$$exprbool : \mathcal{E}xpr \rightarrow \mathcal{E}xprbool$$

$$exprnot : \mathcal{E}xpr \rightarrow \mathcal{E}xprbool$$

A primitiva *exprbool* guarda uma condição advinda de um `THEN` e *exprnot*, por um `ELSE`.

Seja l uma lista de naturais e e uma expressão, tal que $VP(e)(l)$. Define-se a função que extrai as condições de caminho $\Pi : \mathcal{E}xpr \times [\mathbb{N}] \rightarrow [\mathcal{E}xprbool]$ como:

$$
\begin{aligned}
\Pi(e,l) \;:=\; & \textbf{CASE } l \textbf{ OF} \\
& [n : l_1] : \; (\textbf{IF } SA(e,l_1) \neq ite(e_1,e_2,e_3) \;\vee\; n = 0 \\
& \qquad \textbf{THEN } [] \\
& \qquad \textbf{ELSE IF } n = 1 \\
& \qquad \textbf{THEN } [exprbool(SA(e,l_1))] \\
& \qquad \textbf{ELSE } [exprnot(SA(e,l_1))]) + + \Pi(e,l_1); \\
& [] : []
\end{aligned}
$$

Como exemplo de condições de caminho temos:

- $\Pi(P_{er}(gcd), [1,0]) = [exprbool(Z_{zero}(vr))]$
- $\Pi(P_{er}(gcd), [0,1,0]) = [exprbool(Z_{zero}(vr))]$
- $\Pi(P_{er}(gcd), [2,2,0]) = [exprnot(G_{eq}(vr)), exprnot(Z_{zero}(vr))]$

Contextos de chamado são registros que contém informações sobre uma chamada recursiva de uma expressão, como o caminho até a recursão, o seu argumento e as condições de caminho válido até ela. Seja o registro

$$cc : \langle path : [\mathbb{N}],\ rec\_expr : \{a : \mathcal{E}xpr | a = rec(b)\},\ cnds : [\mathcal{E}xprbool] \rangle$$

um contexto de chamada da expressão e. Ele é dito contexto de chamado válido quando $path$ é um caminho válido de e, rec_expr é subtermo de e acessado pelo $path$ e $cnds$ são as condições de caminho também acessados por $path$. Definindo o tipo

$$\mathcal{E}xpr\_cc := \langle path : [\mathbb{N}],\ rec\_expr : \{a : \mathcal{E}xpr | a = rec(b)\},\ cnds : [\mathcal{E}xprbool] \rangle$$

. O predicado $Valid\_cc : \mathcal{E}xpr \to \mathcal{E}xpr\_cc \to \texttt{bool}$ expressa os contextos de chamado válidos:

$$
\begin{aligned}
Valid\_cc(e)(cc) \;:=\; & VP(e)(path(cc)) \;\wedge \\
& rec\_expr(cc) = SA(e, path(cc)) \;\wedge \\
& cnds(cc) = \Pi(e, path(cc))
\end{aligned}
$$

Acima, o acesso aos dados do registro é feito pelas primitivas que têm como entrada um contexto de chamada: $path$, que retorna uma lista de naturais, rec_expr, que retorna uma expressão recursiva e $cnds$, que retorna uma lista de condições.

Para poder avaliar as condições de caminho até o chamado recursivo, e assim, inferir se seu argumento é menor que o argumento chamado, cria-se o avaliador de condições $EC : \mathcal{E}xpr \times [\mathcal{E}xprbool] \times \mathcal{V}al \to \texttt{bool}$.

$$
\begin{aligned}
EC(e, cnds, \beta) \;:=\; & \textbf{CASES}\ cnds\ \textbf{OF} \\
& [] : \ \textbf{true}; \\
& [a : l] : \ (\textbf{CASES}\ a\ \textbf{OF} \\
& \quad exprbool(c) : \varepsilon(c, e, \beta, \nu) \ \wedge\ \nu \\
& \quad exprnot(c) : \varepsilon(c, e, \beta, \nu) \ \wedge\ \textbf{NOT}\ \nu) \wedge \\
& \quad EC(e, l, \beta)
\end{aligned}
$$

Em $EC(e, cnds, \beta)$, avalia-se se conjunção das condições em *cnds* é verdadeira considerando e como expressão recursiva e β como argumento das condições.

Para avaliar se um valor em $\mathcal{V}al$ é menor que outro, usa-se um mapeamento μ em um espaço de medidas $\mathcal{MT}$ onde há uma relação bem fundada $\prec$ entre seus elementos. A relação $\prec$ ser bem fundada é definido como:

$$
\forall\ P : \mathcal{MT} \to \texttt{bool} : ((\exists\ e : P) \Longrightarrow \exists\ (m : P),\ \forall (a : P) : \neg(a \prec m))
$$

Ou seja, não existe sequência infinita decrescente de elementos ordenados por $\prec$:

$$
\overbrace{a_n \succ a_{n-1} \succ \ldots \succ a_3 \succ a_2 \succ a_1 \succ m}^{\text{quantidade finita de } \mathcal{MT} \text{ elementos}}
$$

Onde $\succ$ é a relação conversa de $\prec$.

Logo, usa-se a definição de terminação $T_\zeta : \mathcal{E}xpr \to \texttt{bool}$:

$$
\begin{aligned}
T_\zeta(e_f) \;:=\; & \exists\ \mu,\ \forall\ (\beta,\ cc : Valid\_cc(e_f),\ \nu) : \\
& \varepsilon(get\_arg(rec\_expr(cc)), e_f, \beta, \nu) \wedge \\
& EC(e_f, cnds(cc), \beta) \Longrightarrow \mu(\nu) \prec \mu(\beta)
\end{aligned}
$$

Como dito anteriormente, essa noção de terminação está implementada em diversos assitentes de prova incluindo PVS. Em PVS, quando se especifica um algoritmo recursivo é preciso escrever uma função em um espaço de medidas e uma relação bem fundada, ou apenas uma função que toma os argumentos do algoritmo recursivo e retorna um natural de tal forma que decresça a cada chamado recursivo. Nisso, PVS gera Condições de Correção de Tipos (*Type Correctness Conditions* ou TCC's) que são propriedades relacionadas aos tipos de entrada e saída de funções ou terminação de funções. Logo, T_ζ também é chamado de TCC terminação.

Acima *get_arg* toma uma expressão cuja raiz é recursiva e retorna a expressão do argumento. Ainda acima, μ é uma função que toma um valor $\mathcal{V}al$ e devolve um valor no espaço de medidas $\mathcal{MT}$.

Como exemplo, pode-se avaliar terminação de *gcd* definindo como espaço de medidas $\mathcal{MT} := \mathbb{N}$, $\mathcal{V}al := \mathbb{N} \times \mathbb{N}$ e também definindo a relação $\prec := <$ dos naturais:

$$\begin{aligned} T_\zeta(gcd) \overset{\text{expandido em}}{=} \; & \exists\, \mu,\ \forall\, (\beta,\ cc : Valid\_cc(gcd),\ \nu) : \\ & \varepsilon(get\_arg(rec\_expr(cc)), gcd, \beta, \nu) \wedge \\ & EC(gcd, cnds(cc), \beta) \Longrightarrow \mu(\nu) < \mu(\beta) \end{aligned}$$

Para mostrar a propriedade acima basta encontrar uma função de medida adequada: $\mu(m, n) := n + 2m$:

$$\begin{aligned} & \forall\, (\beta,\ cc : Valid\_cc(gcd),\ \nu) : \\ & \varepsilon(get\_arg(rec\_expr(cc)), gcd, \beta, \nu) \wedge \\ & EC(gcd, cnds(cc), \beta) \Longrightarrow \mu(\nu) < \mu(\beta) \end{aligned}$$

No caso acima, tem que valer para todo argumento β, chamado de contexto válido de *gcd*, *cc* e tupla de naturais ν. As entidades β e ν são infinitas, uma vez que tem-se infinitas possibilidades de tuplas de naturais para β e, portanto, infinitas possibilidades de avaliação ν para $\varepsilon(get\_arg(rec\_expr(cc)), gcd, \beta, \nu)$, mas os chamados de contexto válidos *cc* são finitos. Como possibilidade temos:

1. $$\left(\begin{array}{c} path := [1, 2], \\ rec\_expr := rec(S_{ub}(vr)), \\ cnds := [exprbool(G_{eq}(vr)), exprnot(Z_{ero}(vr))] \end{array} \right)$$

2. $$\left(\begin{array}{c} path := [2, 2], \\ rec\_expr := rec(P_{er}(vr)), \\ cnds := [exprnot(G_{eq}(vr)), exprnot(Z_{ero}(vr))] \end{array} \right)$$

Como ν e *beta* são tuplas de naturais, considera-se $\nu := (\nu_1, \nu_2)$ e $\beta := (b_1, b_2)$ quaisquer. Na primeira possibilidade, considerando simplificações, tem-se:

$$\begin{aligned} & \varepsilon(S_{ub}(vr), gcd, (b_1, b_2), (\nu_1, \nu_2)) \wedge \\ & EC(gcd, [exprbool(G_{eq}(vr)), exprnot(Z_{ero}(vr))], (b_1, b_2)) \Longrightarrow \mu((\nu_1, \nu_2)) < \mu((b_1, b_2)) \end{aligned}$$

Expandindo e simplificando as definições de μ, EC e ε tem-se:

$$\begin{aligned} & (\nu_1, \nu_2) = (b_1, b_2 - b_1) \wedge \\ & b_2 \leq b_1 \wedge \neg(b_1 = 0 \vee b_2 = 0) \Longrightarrow \nu_1 + 2\nu_2 < b_1 + 2b_2 \end{aligned}$$

Decompondo a iguadade $(\nu_1, \nu_2) = (b_1, b_2 - b_1)$ em $\nu_1 = b_1$ e $\nu_2 = b_2 - b_1$, substituindo ν_1 e ν_2 pelos seus iguais e $\neg(b_1 = 0 \vee b_2 = 0)$ por uma fómula equivalente tem-se:

$$b_2 \leq b_1 \wedge b_1 \neq 0 \wedge b_2 \neq 0 \Longrightarrow b_1 + 2(b_2 - b_1) < b_1 + 2b_2$$

A inequação acima é verdadeira pois, partindo do princípio que $b_1 \neq 0$ é verdadeira então pode-se concluir que $-b_1 < b_1$, $2b_2 - b_1 < b_1 + 2b_2$, $b_1 + 2b_2 - 2b_1 < b_1 + 2b_2$, $b_1 + 2(b_2 - b_1) < b_1 + 2b_2$.

Na segunda possibilidade, após simplificações, tem-se

$$\begin{array}{l} \varepsilon(P_{er}(vr), gcd, (b_1, b_2), (\nu_1, \nu_2)) \wedge \\ EC(gcd, [exprnot(G_{eq}(vr)), exprnot(Z_{ero}(vr))], (b_1, b_2)) \Longrightarrow \mu((\nu_1, \nu_2)) < \mu((b_1, b_2)) \end{array}$$

Expandindo e simplificando as definições de μ, EC e ε tem-se:

$$\begin{array}{l} (\nu_1, \nu_2) = (b_2, b_1) \wedge \\ \neg b_2 \leq b_1 \wedge \neg(b_1 = 0 \vee b_2 = 0) \Longrightarrow \nu_1 + 2\nu_2 < b_1 + 2b_2 \end{array}$$

Que é equivalente a:

$$\begin{array}{l} (\nu_1, \nu_2) = (b_2, b_1) \wedge \\ b_2 > b_1 \wedge b_1 \neq 0 \wedge b_2 \neq 0 \Longrightarrow \nu_1 + 2\nu_2 < b_1 + 2b_2 \end{array}$$

Decompondo a igualdade $(\nu_1, \nu_2) = (b_2, b_1)$ em $\nu_1 = b_2$ e $\nu_2 = b_1$ e substituindo ν_1 e ν_2 pelos seus iguais tem-se:

$$b_2 > b_1 \wedge b_1 \neq 0 \wedge b_2 \neq 0 \Longrightarrow b_2 + 2b_1 < b_1 + 2b_2$$

Que é verdadeiro, pois partindo do princípio que $b_1 < b_2$ conclui-se que $2b_1 - b_1 < 2b_2 - b_2$, $2b_1 + b_2 < 2b_2 + b_1$.

Logo, a parada do algoritmo *gcd* está relacionada com o decrescimento do resultado da função μ aplicada aos argumentos a cada chamado recursivo:

$$\begin{array}{ccccccccc} gcd(a_1, b_1) & \overset{chama}{\longrightarrow} & gcd(a_2, b_2) & \overset{chama}{\longrightarrow} & gcd(a_3, a_3) & \overset{chama}{\longrightarrow} & \cdots & \overset{chama}{\longrightarrow} & gcd(a_n, b_n) \\ \downarrow \mu & & \downarrow \mu & & \downarrow \mu & & \cdots & & \downarrow \mu \\ a_1 + 2b_1 & > & a_2 + 2b_2 & > & a_3 + 2b_3 & > & \cdots & > & a_n + 2b_n \end{array}$$

Como cada $a_i + 2b_i$ é um número natural e $>$ é uma relação bem fundada, então não existe uma sequência infinita de chamados de gcd, pois isso implicaria que existe uma sequência infinita de naturais decrescentes.

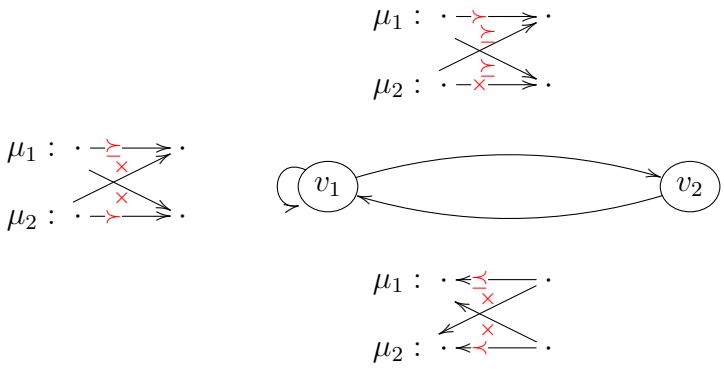


Figura 3.1: grafo de contexto de chamado de gcd

3.4 Outros Critérios de terminação

Conforme dito anteriormente, existem outras noções de terminação também logicamente equivalentes as noções anteriores. Dentre as formalizadas, uma é chamada grafos com matrizes de medida ou MWG [4] e outra é chamada grafos de contexto de chamado ou CCG [17]. Ambas implementam o princípio da mudança de tamanho [16].

Em CCG usam-se grafos dirigdos para representar possíveis caminhos de chamados recursivos. Nesses grafos, cada vértice representa um contexto de chamado e cada aresta, um possível chamado entre os contextos. Adicionalmente, tem-se uma família de medidas $\mu_1, \cdots, \mu_k$. que toma o tipo de argumentos da função analisada e vai a um espaço de medidas que pode ser comparadas por uma ordem bem fundada. Cada aresta é etiquetada conforme a comparação entre diferentes medidas entre os parâmetros da função e os argumentos do chamado recursivo com símbolo de decréscimo se a combinação de medidas leva a um decrescimento, com um símbolo de decrescimento com igualdade caso existe a possibilidade de que a combinação entre as medidas permaneça igual ou decresça ou com um símbolo de indefinido caso nem sempre decresça ou não se sabe. Depois se analisa se em qualquer circuito desse grafo há um decrescimento geral dos argumentos. A figura 3.1 contém o grafo de chamado de contexto com as seguintes medidas $\mu_1(m,n) = m$ e $\mu_2(m,n) = n$ e os seguintes vértices:

$$v_1 = \langle (m,n), \neg(m = 0 \vee n = 0) \wedge n \geq m, (m, n-m) \rangle$$

$$v_2 = \langle (m,n), \neg(m = 0 \vee n = 0) \wedge m > n, (n, m) \rangle$$

Cada vértice do grafo contém uma tripla, onde a primeira parte são os argumentos de entrada, a segunda, as condições de uma chamada recursiva e a terceira, os argumentos do chamado recursivo.

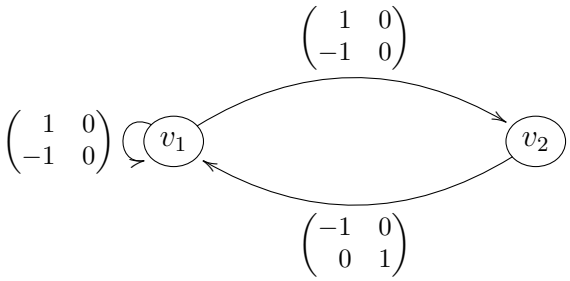


Figura 3.2: grafo com matrizes de medida de gcd

MWG é uma abstração de CCG. Usa-se o mesmo grafo com a diferença que cada aresta é etiquetada com uma matriz com pesos de medida. Cada peso pode ser 1, 0 e -1. Cada i-ésima e j-ésima coluna é uma comparação das imagens da i-ésima e j-ésima medida, a primeira sobre os parâmetros da função e a segunda sobre os argumentos do chamado recursivo do contexto de chamado. Se há um decrescimento o peso é 1. Se há um decrescimento ou igualdade o peso é 0. Se for indefinido ou nem sempre há decrescimento o peso é -1. Para cada circuito faz-se uma operação de multiplicação de matrizes que estão nas arestas. Se sempre o resultado for uma matriz com pelo menos peso um peso 1 na diagonal principal, ou seja, a matriz é positiva, então a função verificada termina. Na figura 3.2 tem-se o grafo com matrizes de medida com as medidas $\mu_1(m, n) = m$ e $\mu_2(m, n) = n$

Capítulo 4

Formalização da equivalência entre noções de terminação

4.1 Equivalencia entre as noções de terminação semântica

Formalizou-se a a equivalência entre as noções de terminação semântica abaixo:

$$\forall e_f : \mathcal{E}xpr : \ T_\varepsilon(e_f) \equiv T_\chi(e_f)$$

O teorema acima está especificado no arquivo pvs0_expr que pode ser visto na figura 1.1 conforme o seguinte lema:

```
eval_expr_terminates:  LEMMA
FORALL (expr,body:PVS0Expr):
eval_expr_termination(false_val,eval_op1,eval_op2)(expr,body)
IFF terminates_expr(false_val,eval_op1,eval_op2)(expr,body)
```

Para mostrar a equivalência foi necessário provar algumas propriedades:

1. $\forall (m, n) : m \geq n$:

   $\chi(n, e, e_f, \beta) \in \mathcal{V}al \ \wedge \ \chi(n, e, e_f, \beta) = \nu \ \Rightarrow \ \chi(m, e, e_f, \beta) = \nu$

2. $\varepsilon(e, e_f, \beta, \nu) \ \Rightarrow \ \exists\, m \in \mathbb{N} : \ \chi(m, e, e_f, \beta) \in \mathcal{V}al \ \wedge \ \chi(m, e, e_f, \beta) = \nu$

3. $\varepsilon(e, e_f, \beta, \nu_1) \ \wedge \ \varepsilon(e, e_f, \beta, \nu_2) \ \Rightarrow \ \nu_1 = \nu_2$

A primeira propriedade acima provem da definição de χ, a segunda é obtida por uma indução na estrutura de ε, e a terceira é consequência das anteriores.

A segunda propriedade é provada da seguinte forma:

Lema 1.

$$\forall(e, e_f, \beta, \nu) : \varepsilon(e, e_f, \beta, \nu) \;\Rightarrow\; \exists\, m \in \mathbb{N} :\; \chi(m, e, e_f, \beta) \in \mathcal{V}al \;\wedge\; \chi(m, e, e_f, \beta) = \nu$$

Demonstração. Indução na estrutura de ε.

- Caso e é um símbolo de constante $cnst$. Logo

  $$\forall(e_f, \beta, \nu) : \varepsilon(cnst, e_f, \beta, \nu) \;\Rightarrow\; \exists\, m \in \mathbb{N} :\; \chi(m, cnst, e_f, \beta) \in \mathcal{V}al \wedge \chi(m, cnst, e_f, \beta) = \nu$$

  Expandindo ε e χ em suas definições:

  $$\begin{aligned} &\forall(e_f, \beta, \nu) : cnst^{\mathfrak{I}} = \nu \;\Rightarrow \\ \exists\, m \in \mathbb{N} :\; &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } cnst^{\mathfrak{I}}) \in \mathcal{V}al \wedge \\ &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } cnst^{\mathfrak{I}}) = \nu \end{aligned}$$

  Para e_f, β e ν qualquer e supondo $cnst^{\mathfrak{I}} = \nu$:

  $$\begin{aligned} \exists\, m \in \mathbb{N} :\; &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } cnst^{\mathfrak{I}}) \in \mathcal{V}al \wedge \\ &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } cnst^{\mathfrak{I}}) = \nu \end{aligned}$$

  Basta tomar $m := 1$ para que a assertiva acima valer:

  $$\begin{aligned} &cnst^{\mathfrak{I}} \in \mathcal{V}al \wedge \\ &\quad cnst^{\mathfrak{I}} = \nu \end{aligned}$$

- Caso e é símbolo de variável vr:

  $$\forall(e_f, \beta, \nu) : \varepsilon(vr, e_f, \beta, \nu) \;\Rightarrow\; \exists\, m \in \mathbb{N} :\; \chi(m, vr, e_f, \beta) \in \mathcal{V}al \;\wedge\; \chi(m, vr, e_f, \beta) = \nu$$

  Expandindo ε e χ em suas definições:

  $$\begin{aligned} &\forall(e_f, \beta, \nu) : \beta = \nu \;\Rightarrow \\ \exists\, m \in \mathbb{N} :\; &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } \beta \in \mathcal{V}al \wedge \\ &(\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } \beta) = \nu \end{aligned}$$

  Para e_f, β e ν qualquer e supondo $\beta = \nu$:

$$\exists\, m \in \mathbb{N} :\ (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } \beta) \in \mathcal{V}al \wedge \\ (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE } \beta) = \nu$$

Basta tomar $m := 1$ para que a assertiva acima valer:

$$\beta \in \mathcal{V}al \wedge \\ \beta = \nu$$

- Caso e seja um operador unário: $e := op1(e_1)$

  Por hipótese de indução vale:

$$\forall(e_f, \beta, \nu) : \varepsilon(e_1, e_f, \beta, \nu) \ \Rightarrow\ \exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, e_1, e_f, \beta) = \nu$$

  Tem-se a assertiva:

$$\forall(e_f, \beta, \nu) : \varepsilon(op1(e_1), e_f, \beta, \nu) \ \Rightarrow \\ \exists\, m \in \mathbb{N} :\ \chi(m, op1(e_1), e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, op1(e_1), e_f, \beta) = \nu$$

  Expandindo ε e χ:

$$\begin{aligned} \forall(e_f, \beta, \nu) : \exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \nu = op1^{\mathfrak{I}}(\nu_1) \ \Rightarrow \\ \exists\, m \in \mathbb{N} :\quad & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta)) \textbf{ ELSE } \diamondsuit) \in \mathcal{V}al \wedge \\ & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta)) \textbf{ ELSE } \diamondsuit) = \nu \end{aligned}$$

  Para e_f, β e ν qualquer e supondo $\exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \nu = op1^{\mathfrak{I}}(\nu_1)$

$$\begin{aligned} \exists\, m \in \mathbb{N} :\quad & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta) \textbf{ ELSE } \diamondsuit) \in \mathcal{V}al \wedge \\ & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta)) \textbf{ ELSE } \diamondsuit) = \nu \end{aligned}$$

Por hipótese de indução vale $\exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, e_1, e_f, \beta) = \nu_1$. Logo, a assertiva anterior pode ser simplificada para:

$$\begin{aligned}\exists\, m \in \mathbb{N} :\quad & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta)) \in \mathcal{V}al\ \wedge \\ & op1^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta)) = \nu\end{aligned}$$

Caso e é operador binário: $e := op2(e_1, e_2)$

Por hipótese de indução valem:

$$\forall(e_f, \beta, \nu) : \varepsilon(e_1, e_f, \beta, \nu)\ \Rightarrow\ \exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, e_1, e_f, \beta) = \nu$$

$$\forall(e_f, \beta, \nu) : \varepsilon(e_2, e_f, \beta, \nu)\ \Rightarrow\ \exists\, m \in \mathbb{N} :\ \chi(m, e_2, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, e_2, e_f, \beta) = \nu$$

Tem-se que provar a assertiva:

$$\begin{aligned}&\forall(e_f, \beta, \nu) : \varepsilon(op2(e_1, e_2), e_f, \beta, \nu)\ \Rightarrow \\ &\exists\, m \in \mathbb{N} :\ \chi(m, op2(e_1, e_2), e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, op2(e_1, e_2), e_f, \beta) = \nu\end{aligned}$$

Expandindo ε e χ:

$$\begin{aligned}&\forall(e_f, \beta, \nu) : \exists \nu_1, \nu_2 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \varepsilon(e_2, e_f, \beta, \nu_2) \wedge \nu = op2^{\mathfrak{I}}(\nu_1, \nu_2)\ \Rightarrow \\ &\exists\, m \in \mathbb{N} :\quad (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ &\qquad \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ &\qquad op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) \in \mathcal{V}al\ \wedge \\ &\qquad (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ &\qquad \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ &\qquad op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) = \nu\end{aligned}$$

Para e_f, β e ν qualquer e supondo $\exists \nu_1, \nu_2 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \varepsilon(e_2, e_f, \beta, \nu_2) \wedge \nu = op2^{\mathfrak{I}}(\nu_1, \nu_2)$:

$$\begin{aligned}\exists\, m \in \mathbb{N} :\quad & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) \in \mathcal{V}al\ \wedge \\ & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) = \nu\end{aligned}$$

Simplificando as hipóteses de indução:

$$\exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, e_1, e_f, \beta) = \nu_1$$

$$\exists\, m \in \mathbb{N} :\ \chi(m, e_2, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, e_2, e_f, \beta) = \nu_2$$

Simplificando os quantificadores existenciais:

$$\chi(h, e_1, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(h, e_1, e_f, \beta) = \nu_1$$

$$\chi(d, e_2, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(d, e_2, e_f, \beta) = \nu_2$$

Entre os números naturais h e d, escolhe-se o maior deles. Chame de t. Também valerá:

$$\chi(t, e_1, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(t, e_1, e_f, \beta) = \nu_1$$

$$\chi(t, e_2, e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(t, e_2, e_f, \beta) = \nu_2$$

É preciso provar:

$$\begin{aligned}\exists\, m \in \mathbb{N} :\quad & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) \in \mathcal{V}al \wedge \\ & (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\ & \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(m, e_2, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\ & op2^{\mathfrak{I}}(\chi(m, e_1, e_f, \beta), \chi(m, e_2, e_f, \beta)) \textbf{ ELSE } \diamondsuit) = \nu\end{aligned}$$

Escolhendo $m := t$ e simplificando:

$$\begin{aligned}& op2^{\mathfrak{I}}(\chi(t, e_1, e_f, \beta), \chi(t, e_2, e_f, \beta)) \in \mathcal{V}al \wedge \\ & op2^{\mathfrak{I}}(\chi(t, e_1, e_f, \beta), \chi(t, e_2, e_f, \beta)) = \nu\end{aligned}$$

Caso e seja o operador de `IF THEN ELSE`: $e := ite(e_1, e_2, e_3)$. Tem-se

$$\begin{aligned}& \forall (e_f, \beta, \nu) : \varepsilon(ite(e_1, e_2, e_3), e_f, \beta, \nu)\ \Rightarrow \\ & \exists\, m \in \mathbb{N} :\ \chi(m, ite(e_1, e_2, e_3), e_f, \beta) \in \mathcal{V}al\ \wedge\ \chi(m, ite(e_1, e_2, e_3), e_f, \beta) = \nu\end{aligned}$$

Expandindo ε e χ:

$$
\begin{aligned}
&\forall(e_f, \beta, \nu) : (\exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \\
&\textbf{IF } \nu_1 \textbf{ THEN } \varepsilon(e_2, e_f, \beta, \nu) \textbf{ ELSE } \varepsilon(e_2, e_f, \beta, \nu)) \ \Rightarrow \\
&\exists\, m \in \mathbb{N} : \quad (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\
&\qquad\qquad \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
&\qquad\qquad\quad \textbf{IF } \chi(m, e_1, e_f, \beta) \textbf{ THEN } \chi(m, e_2, e_f, \beta) \\
&\qquad\qquad\quad \textbf{ELSE } \chi(m, e_3, e_f, \beta) \\
&\qquad\qquad \textbf{ELSE } \diamondsuit) \in \mathcal{V}al \\
&\qquad\qquad (\textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \textbf{ ELSE} \\
&\qquad\qquad \textbf{IF } \chi(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
&\qquad\qquad\quad \textbf{IF } \chi(m, e_1, e_f, \beta) \textbf{ THEN } \chi(m, e_2, e_f, \beta) \\
&\qquad\qquad\quad \textbf{ELSE } \chi(m, e_3, e_f, \beta) \\
&\qquad\qquad \textbf{ELSE } \diamondsuit) = \nu
\end{aligned}
$$

Para e_f, β e ν qualquer e supondo:

$$\exists \nu_1 : \varepsilon(e_1, e_f, \beta, \nu_1) \wedge \textbf{IF } \nu_1 \textbf{ THEN } \varepsilon(e_2, e_f, \beta, \nu) \textbf{ ELSE } \varepsilon(e_2, e_f, \beta, \nu)$$

Por hipótese de indução vale:

$$\forall(e_f, \beta, \nu) : \varepsilon(e_1, e_f, \beta, \nu) \ \Rightarrow\ \exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, e_1, e_f, \beta) = \nu$$

Portanto:

$$\exists\, m \in \mathbb{N} :\ \chi(m, e_1, e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, e_1, e_f, \beta) = \nu_1$$

Se ν_1 for verdadeiro então vale $\varepsilon(e_2, e_f, \beta, \nu)$. Logo, por hipótese de indução:

$$\exists\, m \in \mathbb{N} :\ \chi(m, e_2, e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, e_2, e_f, \beta) = \nu$$

Se ν_1 for falso então vale $\varepsilon(e_3, e_f, \beta, \nu)$. Logo, por hipótese de indução:

$$\exists\, m \in \mathbb{N} :\ \chi(m, e_3, e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, e_3, e_f, \beta) = \nu$$

Caso e seja a recursão: $e := rec(e_1)$.

É preciso provar:

$$
\begin{aligned}
&\forall(e_f, \beta, \nu) : \varepsilon(rec(e_1), e_f, \beta, \nu) \ \Rightarrow \\
&\exists\, m \in \mathbb{N} :\ \chi(m, rec(e_1), e_f, \beta) \in \mathcal{V}al \ \wedge\ \chi(m, rec(e_1), e_f, \beta) = \nu
\end{aligned}
$$

Expandindo ε e χ:

$$
\begin{aligned}
&\forall(e_f,\beta,\nu):\exists\beta':\varepsilon(e_1,e_f,\beta,\beta')\wedge\varepsilon(e_f,e_f,\beta',\nu)\ \Rightarrow\\
&\exists\, m\in\mathbb{N}:\ \ (\textbf{IF } m=0 \textbf{ THEN } \diamondsuit\\
&\qquad\qquad \textbf{ELSE IF } \chi(m,e_1,e_f,\beta)\neq\diamondsuit \textbf{ THEN}\\
&\qquad\qquad\ \chi(m-1,e_f,e_f,\chi(m,e_1,e_f,\beta))\\
&\qquad\qquad \textbf{ELSE } \diamondsuit)\in\mathcal{V}al\wedge\\
&\qquad\qquad (\textbf{IF } m=0 \textbf{ THEN } \diamondsuit\\
&\qquad\qquad \textbf{ELSE IF } \chi(m,e_1,e_f,\beta)\neq\diamondsuit \textbf{ THEN}\\
&\qquad\qquad\ \chi(m-1,e_f,e_f,\chi(m,e_1,e_f,\beta))\\
&\qquad\qquad \textbf{ELSE } \diamondsuit)=\nu
\end{aligned}
$$

Para e_f, β, ν qualquer e supondo $\exists\beta':\varepsilon(e_1,e_f,\beta,\beta')\wedge\varepsilon(e_f,e_f,\beta',\nu)$, por hipótese de indução tem-se:

$$\forall(e_f,\beta,\nu):\varepsilon(e_1,e_f,\beta,\nu)\ \Rightarrow\ \exists\, m\in\mathbb{N}:\ \chi(m,e_1,e_f,\beta)\in\mathcal{V}al\ \wedge\ \chi(m,e_1,e_f,\beta)=\nu$$

$$\forall(e_f,\beta,\nu):\varepsilon(e_f,e_f,\beta,\nu)\ \Rightarrow\ \exists\, m\in\mathbb{N}:\ \chi(m,e_f,e_f,\beta)\in\mathcal{V}al\ \wedge\ \chi(m,e_f,e_f,\beta)=\nu$$

Que simplificando fica:

$$\exists\, m\in\mathbb{N}:\ \chi(m,e_1,e_f,\beta)\in\mathcal{V}al\ \wedge\ \chi(m,e_1,e_f,\beta)=\beta'$$

$$\exists\, m\in\mathbb{N}:\ \chi(m,e_f,e_f,\beta')\in\mathcal{V}al\ \wedge\ \chi(m,e_f,e_f,\beta')=\nu$$

Eliminando os quantificadores existenciais:

$$\chi(n1,e_1,e_f,\beta)\in\mathcal{V}al\ \wedge\ \chi(n1,e_1,e_f,\beta)=\beta'$$

$$\chi(n2,e_f,e_f,\beta')\in\mathcal{V}al\ \wedge\ \chi(n2,e_f,e_f,\beta')=\nu$$

O valor $n1+n2+1$ é maior que $n1$ e $n2$ e pode ser usado na avaliação de uma expressão recursiva. Portanto vale:

$$\chi(n1+n2+1,e_1,e_f,\beta)\in\mathcal{V}al\ \wedge\ \chi(n1+n2+1,e_1,e_f,\beta)=\beta'$$

$$\chi(n1+n2,e_f,e_f,\beta')\in\mathcal{V}al\ \wedge\ \chi(n1+n2,e_f,e_f,\beta')=\nu$$

E também vale:

$$
\begin{array}{l}
(\textbf{IF } n1 + n2 + 1 = 0 \textbf{ THEN } \diamondsuit \\
\textbf{ELSE IF } \chi(n1 + n2 + 1, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
\quad \chi(n1 + n2, e_f, e_f, \chi(n1 + n2 + 1, e_1, e_f, \beta)) \\
\textbf{ELSE } \diamondsuit) \in \mathcal{V}al \wedge \\
(\textbf{IF } n1 + n2 + 1 = 0 \textbf{ THEN } \diamondsuit \\
\textbf{ELSE IF } \chi(n1 + n2 + 1, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
\quad \chi(n1 + n2, e_f, e_f, \chi(n1 + n2 + 1, e_1, e_f, \beta)) \\
\textbf{ELSE } \diamondsuit) = \nu
\end{array}
$$

□

Para mostrar $T_\varepsilon(e_f) \equiv T_\chi(e_f)$ tem-se os seguintes passos:

- $T_\varepsilon(e_f) \Rightarrow T_\chi(e_f)$
  - Considerando que para todo β existe um valor ν onde $\varepsilon(e_f, e_f, \beta, \nu)$
  - Pela propriedade 2 acima há um m tal que : $\chi(m, e_f, e_f, \beta) \in \mathcal{V}al \wedge \chi(m, e_f, e_f, \beta) = \nu$
- $T_\varepsilon(e_f) \Leftarrow T_\chi(e_f)$
  - Considerando que há um m tal que $\chi(m, e_f, e_f, \beta) \in \mathcal{V}al \ \wedge \ \varepsilon(e_f, e_f, \beta, \nu)$
  - Então $\varepsilon(e_f, e_f, \beta, \nu)$

4.2 Equivalência entre TCC terminação e terminação semântica

Uma das contribuições deste trabalho foi a formalização de lemas auxiliares para equivalência entre T_ζ e as noções semânticas de terminação. Considerando a equivalência $T_\varepsilon(e_f) \equiv T_\chi(e_f)$ como válida (trabalho concluído por Muñoz com contribuições de Avelar e Ayala-Rincón em PVS e escrito na tese de doutorado por Avelar em [4]), será mostrado como foi feita essa demonstração.

O lema abaixo diz que se existe terminação semântica então existe TCC terminação.

Lema 2.

$$\forall e_f : \mathcal{E}xpr : T_\varepsilon(e_f) \Rightarrow T_\zeta(e_f)$$

Este lema está no arquivo measure_termination conforme a figura 1.1.

Demonstração. Deseja-se provar, para uma expressão qualquer e_f, que

$$\forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon(e_f, e_f, \beta, \nu)$$
$$\Rightarrow$$
$$\exists\ \mu,\ \forall\ (\beta,\ cc : Valid\_cc(e_f),\ \nu) :$$
$$\varepsilon(get\_arg(rec\_expr(cc)), e_f, \beta, \nu) \wedge$$
$$EC(e_f, cnds(cc), \beta) \Longrightarrow \mu(\nu) \prec \mu(\beta)$$

Então suponha $T_\varepsilon(e_f)$ para uma expressão qualquer e_f. Por definição, para todo mapeamento β existe um valor ν tal que $\varepsilon(e_f, e_f, \beta, \nu)$. Por equivalência entre T_ε e T_χ, temos que existe um valor n tal que $\chi(n, e_f, e_f, \beta) = \nu_1 \wedge \nu_1 \neq \diamondsuit \wedge \varepsilon(e_f, e_f, \beta, \nu_1)$. Se existe esse valor, existe também um valor mínimo para avaliar subexpressões de e_f. Esse valor é a altura da árvore de chamadas recursivas. Assim, pode-se definir uma função $\mu(e_f)(i) := min\{m | \chi(m, e_f, e_f, i) \neq \diamondsuit\}$, ou seja, a altura da árvore de recursão da função representada por e_f avaliando como argumento i. Em PVS, ficou assim:

```
mu(pvs0:PVS0)(val:Val|determined?(pvs0,val)):  posnat =
LET S = { n :  nat | LET myv = eval(pvs0)(n)(val) IN some?(myv) } IN
min(S)
```

Nesse caso, precisou-se mostrar propriedades da correção de μ, tais como:

- O conjunto $\{m | \chi(m, e_f, e_f, i) \neq \diamondsuit\}$ não é vazio e, portanto, pode-se tomar o mínimo elemento. Nesse caso, prova-se com hipótese de terminação semântica de e_f.
- $\mu(e_f)(i)$ funciona como um limite para o número de níveis de chamado recursivo. Isso é provado com a própria definição de μ.

Com isso, tem-se a especificação da correção de μ em PVS, que foi uma contribuição deste trabalho:

```
mu_min :  LEMMA
FORALL (pvs0:PVS0,val:Val|determined?(pvs0,val),n:nat):
some?(eval(pvs0)(n)(val)) IMPLIES mu(pvs0)(val) <= n
```

Na especificação acima, é dito que para uma definição de expressão de função terminante `pvs0`, quando avaliado com o argumento `val` e retornando a algum valor para o limite de n recursões possui um número de limites maior que o valor de μ.

Outra especificação de correção está expressa em:

```
mu_terminates :  LEMMA
FORALL (pvs0:PVS0,val:Val|determined?(pvs0,val)):
LET mun = mu(pvs0)(val),
myv = eval(pvs0)(mun)(val) IN
some?(myv)
```

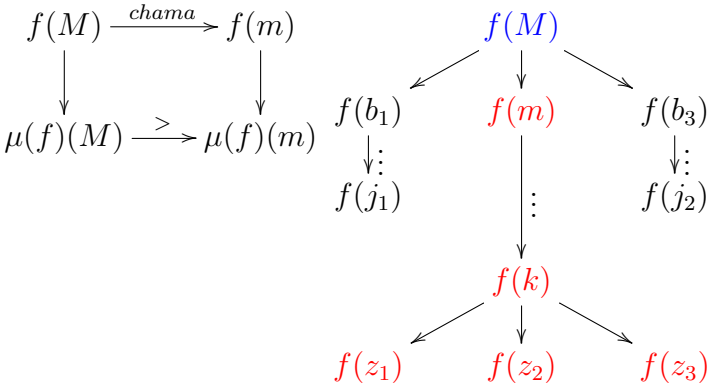
Acima, o valor de μ indicado em `mu` é um limite nos níveis de recursão.

Continuando a prova, pode-se também usar como relação bem fundada a ordem dos naturais: $\prec := <$. Quando se avalia uma função com um argumento e altura de árvore $i+1$ e se chega em uma chamada recursiva (em

$$\varepsilon(get\_arg(rec\_expr(cc)), e_f, \beta, \nu)$$

, cc é o contexto de chamado com a expressão recursiva e ν é o resultado da avaliação de seu argumento), logo após as condições de caminho serem dadas como verdadeiras (ou seja, $EC(e_f, cnds(cc), \beta)$), a altura da subárvore será menor ou igual a i. Assim, $\varepsilon(get\_arg(rec\_expr(cc)), e_f, \beta, \nu) \wedge EC(e_f, cnds(cc), \beta) \implies \mu(e_f)(\nu) < \mu(e_f)(\beta)$ para qualquer contexto de chamado cc de e_f e qualquer argumento proveniente de β.

□



Na figura acima, percebe-se que a função f com argumento M chama recursivamente f com argumento m e a altura de $f(M)$ é maior que $f(m)$. Essa altura é usada para provar o lema anterior.

Esse tipo de propriedade precisou ser demonstrada conforme pode ser visto abaixo. Para se demonstrar o lema abaixo, precisou-se demonstrar outros lemas auxiliares e alguns TCC's. Dentre essas propriedades, formalizadas pelo autor, estão:

- A função `valid_paths` toma uma expressão e retorna uma lista com todos os seus caminhos válidos.
- Dado um conjunto de condições verdadeiras implementado como uma lista, qualquer outro subconjunto implementado como lista terá condições verdadeiras.
- Dados dois caminhos válidos em uma expressão *path* e *subpath* onde *subpath* é um sufixo de *path*, se as condições extraídas de um caminho válido por *path* são verdadeiras, então as condições extraídas por *subpath* são verdadeiras também.

Para o lema acima valer precisou ser formalizado que $\mu(e_f)$ decresce a cada chamado recursivo, conforme o lema abaixo, que diz que o tamanho da árvore de recursão para um chamado recursivo é menor que o tamanho da árvore de recursão total para uma expressão terminante:

Lema 3.

$$\forall(\beta, e_f, e_1, path) : EC(e_f, \Pi(e_f, path), \beta) \wedge SA(e_f, path) = rec(e_1) \wedge T_\varepsilon(e_f)$$
$$\Longrightarrow$$
$$\forall \nu : \varepsilon(e_1, e_f, \beta, \nu) \Longrightarrow \mu(e_f)(\nu) < \mu(e_f)(\beta)$$

Esse lema está no arquivo pvs0_props conforme a figura 1.1.

Demonstração. Suponha que para β, e_f, e_1, $path$ quaisquer vale:

$$EC(e_f, \Pi(e_f, path), \beta) \wedge SA(e_f, path) = rec(e_1) \wedge T_\varepsilon(e_f)$$

Suponha também para ν qualquer:

$$\varepsilon(e_1, e_f, \beta, \nu)$$

Assumindo conforme o lema abaixo:

$$\forall(\beta, n, e, a, e_f) :$$
$$\exists p_1, p_2 : e = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$
$$\Longrightarrow$$
$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta) \text{ e}$$
$$\text{suffix}(b, ab) := \exists k : \ k + +b = ab$$

Instanciando a propriedade como $\beta := \beta$, $n := \mu(e_f)(\beta)$, $e := e_f$, $a := e_1$ e $e_f := e_f$:

$$\exists p_1, p_2 : e_f = SA(e_f, p_1) \wedge rec(e_1) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(\mu(e_f)(\beta), e_f, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$
$$\Longrightarrow$$
$$\chi(\mu(e_f)(\beta), e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(\mu(e_f)(\beta) - 1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(\mu(e_f)(\beta), e_1, e_f, \beta) \text{ e}$$
$$\text{suffix}(b, ab) := \exists k : \ k + +b = ab$$

Instanciando novamente com $p_1 := []$ e $p_2 := path$:

$$\chi(\mu(e_f)(\beta), e_f, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, path)), \beta)$$
$$\Longrightarrow$$
$$\chi(\mu(e_f)(\beta), e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(\mu(e_f)(\beta) - 1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(\mu(e_f)(\beta), e_1, e_f, \beta) \text{ e}$$
$$\text{suffix}(b, ab) := \exists k : \ k + +b = ab$$

Pela própria definição de μ e por $EC(e_f, cnds(\Pi(e_f, path)), \beta)$ tem-se:

$$\chi(\mu(e_f)(\beta), e_1, e_f, \beta) \neq \diamondsuit \wedge \chi(\mu(e_f)(\beta) - 1, e_f, e_f, \chi(\mu(e_f)(\beta), e_1, e_f, \beta)) \neq \diamondsuit$$

Como $\nu = \chi(\mu(e_f)(\beta), e_1, e_f, \beta)$ pois $\varepsilon(e_1, e_f, \beta, \nu)$ então:

$$\chi(\mu(e_f)(\beta) - 1, e_f, e_f, \nu) \neq \diamondsuit$$

Pela definição de μ:
$\mu(e_f)(\nu) \leq \mu(e_f)(\beta) - 1$ □

O lema abaixo diz que em uma expressão que representa uma função, e_f, sempre que se consegue avaliar qualquer subexpressão e, tal que, em e há uma recursão $rec(a)$ ($rec(a)$ é subexpressão de e porque p_1 é sufixo de p_2) com uma árvore de recursão de tamanho menor ou igual a n, e para isso precisou-se avaliar $rec(a)$ (ou seja, o caminho de recursão é verdadeiro conforme está escrito em $EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$), a avaliação de $rec(a)$ será um tamanho menor ou igual a $n - 1$.

Lema 4.

$$\forall(\beta, n, e, a, e_f):$$
$$\exists p_1, p_2 : e = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\mathit{suffix}(p_1, p_2) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$
$$\Longrightarrow$$
$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\mathit{Onde}\ \beta' := \chi(n, a, e_f, \beta)\ e$$
$$\mathit{suffix}(b, ab) := \exists k :\ k + +b = ab$$

Esse lema está em pvs0_props conforme a figura 1.1

Demonstração. Indução na expressão e.

- Caso e é uma constante: $e := cnst$. Então, a condição antes da implicação é falsa. Essa condição será:

$$\exists p_1, p_2 : cnst = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, cnst, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

  Se $cnst = SA(e_f, p_1)$, suffix(p_1, p_2) e $rec(a) = SA(e_f, p_2)$, então $p_2 = p_1$, pois p_2 é um caminho válido de e_f e p_1 é maximal. Assim $cnst = rec(a)$ que é falso. Assim vale:

$$\forall(\beta, n, e, a, e_f) : \mathbf{FALSE}$$
$$\Longrightarrow$$
$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

- Caso e é uma variável: $e := vr$. A prova é a mesma do caso anterior.
- Caso e é um operador unário: $e = op1(e_1)$.

Como hipótese de indução tem-se:

$$\begin{gathered}\forall(\beta, n, a, e_f):\\ \exists p_1, p_2 : e_1 = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge\\ \text{suffix}(p_1, p_2) \wedge\\ \chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge\\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\\ \Longrightarrow\\ \chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit\\ \text{Onde } \beta' := \chi(n, a, e_f, \beta)\end{gathered}$$

Supondo para β, n, a e e_f quaisquer;

$$\begin{gathered}\exists p_1, p_2 : op1(e_1) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge\\ \text{suffix}(p_1, p_2) \wedge\\ \chi(n, op1(e_1), e_f, \beta) \neq \diamondsuit \wedge\\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

Expandindo a definição de χ e substintuindo p_1 por $[0] + +p_1$ tem-se:

$$\begin{gathered}\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge\\ \text{suffix}([0] + +p_1, p_2) \wedge\\ \chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge\\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

No caso acima $[0] + +p_1$ é sufixo de p_2 porque $rec(a)$ é subtermo de $op1(e_1)$ e, portanto, é subtermo de e_1. Pela hipótese de indução vale:

$$\begin{gathered}\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit\\ \text{Onde } \beta' := \chi(n, a, e_f, \beta)\end{gathered}$$

- Caso e é um operador binário: $e = op2(e_1, e_2)$ Assim, deve-se provar:

$$\forall(\beta, n, a, e_f):$$
$$\exists p_1, p_2 : op2(e_1, e_2) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, op2(e_1, e_2), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$
$$\Longrightarrow$$
$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

Para β, n, a e e_f quaisquer supor:

$$\exists p_1, p_2 : op2(e_1, e_2) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, op2(e_1, e_2), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Como $op2(e_1, e_2) = SA(e_f, p_1)$ e suffix(p_1, p_2) tem-se dois casos: suffix$([0] + +p_1, p_2)$ ou suffix$([1] + +p_1, p_2)$. Caso suffix$([0] + +p_1, p_2)$ tem-se:

$$\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([0] + +p_1, p_2) \wedge$$
$$\chi(n, op2(e_1, e_2), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Expandindo χ:

$$\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([0] + +p_1, p_2) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Aplicando a hipótese de indução:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

De maneira semelhante ocorrerá se suffix$([1] + +p_1, p_2)$:

$$\exists p_1, p_2 : e_1 = SA(e_f, [1] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$suffix([1] + +p_1, p_2) \wedge$$
$$\chi(n, op2(e_1, e_2), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Expandindo χ:

$$\exists p_1, p_2 : e_1 = SA(e_f, [1] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([1] + +p_1, p_2) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Aplicando a hipótese de indução:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

- Caso e é `IF THEN ELSE`: $e = ite(e_1, e_2, e_3)$.

  Assim, deve-se provar:

$$\forall(\beta, n, a, e_f) :$$
$$\exists p_1, p_2 : ite(e_1, e_2, e_3) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$
$$\Longrightarrow$$
$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

  Para β, n, a e e_f quaisquer supor:

$$\exists p_1, p_2 : ite(e_1, e_2, e_3) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Há três possibilidades : suffix$([0]++p_1, p_2)$, suffix$([1]++p_1, p_2)$, suffix$([2]++p_1, p_2)$.

Caso suffix$([0] + +p_1, p_2)$, tem-se:

$$\begin{gathered}\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge \\ \text{suffix}([0] + +p_1, p_2) \wedge \\ \chi(n, ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge \\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

Expandindo a definição de χ:

$$\begin{gathered}\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge \\ \text{suffix}([0] + +p_1, p_2) \wedge \\ \chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

Que, por hipótese de indução vale:

$$\begin{gathered}\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit \\ \text{Onde } \beta' := \chi(n, a, e_f, \beta)\end{gathered}$$

Caso suffix$([1] + +p_1, p_2)$, tem-se:

$$\begin{gathered}\exists p_1, p_2 : e_1 = SA(e_f, [1] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge \\ \text{suffix}([1] + +p_1, p_2) \wedge \\ \chi(n, ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge \\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

Por $EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$ e suffix$([1] + +p_1, p_2)$ então $EC(e_f, cnds(\Pi(e_f, [1] + +p_1)), \beta)$ (explicado no lema abaixo) o que significa que $\chi(n, e_1, e_f, \beta) =$ **True**. Logo, expandindo a definição de χ tem-se:

$$\begin{gathered}\exists p_1, p_2 : e_1 = SA(e_f, [1] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge \\ \text{suffix}([1] + +p_1, p_2) \wedge \\ \chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\ EC(e_f, cnds(\Pi(e_f, p_2)), \beta)\end{gathered}$$

Logo, por hipótese de indução vale:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

Caso suffix$([2] + +p_1, p_2)$, tem-se:

$$\exists p_1, p_2 : e_1 = SA(e_f, [2] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([2] + +p_1, p_2) \wedge$$
$$\chi(n, ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Por $EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$ e suffix$([2] + +p_1, p_2)$ então $EC(e_f, cnds(\Pi(e_f, [2] + +p_1)), \beta)$ o que significa que $\chi(n, e_1, e_f, \beta) = \textbf{False}$. Logo, expandindo a definição de χ tem-se:

$$\exists p_1, p_2 : e_1 = SA(e_f, [2] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([2] + +p_1, p_2) \wedge$$
$$\chi(n, e_3, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Logo, por hipótese de indução vale:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

- Caso e é recursão: $e := rec(e_1)$

  Supondo para β, n, a e e_f quaisquer;

$$\exists p_1, p_2 : rec(e_1) = SA(e_f, p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}(p_1, p_2) \wedge$$
$$\chi(n, rec(e_1), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

  Se $p_1 = p_2$,tem-se:

$$\exists p_1 : rec(a) = SA(e_f, p_1) \wedge$$
$$\chi(n, rec(a), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_1)), \beta)$$

Ao expandir χ vale:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

Se $p_1 \neq p_2$ então $[0] + +p_1$ é sufixo de p_2:

$$\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([0] + +p_1, p_2) \wedge$$
$$\chi(n, rec(e_1), e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Se $\chi(n, rec(e_1), e_f, \beta) \neq \diamondsuit$ então $\chi(n, e_1, e_f, \beta) \neq \diamondsuit$:

$$\exists p_1, p_2 : e_1 = SA(e_f, [0] + +p_1) \wedge rec(a) = SA(e_f, p_2) \wedge$$
$$\text{suffix}([0] + +p_1, p_2) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$EC(e_f, cnds(\Pi(e_f, p_2)), \beta)$$

Por hipótese de indução vale:

$$\chi(n, a, e_f, \beta) \neq \diamondsuit \wedge \chi(n-1, e_f, e_f, \beta') \neq \diamondsuit$$
$$\text{Onde } \beta' := \chi(n, a, e_f, \beta)$$

□

Ainda para a demonstração acima, usou-se o lema abaixo, que diz que quando e é uma subexpressão de uma expressão funcional qualquer e_f (aqui expresso em $\exists p : \ e = SA(e_f, p)$), quando se avalia as condições de um caminho *path* de e como verdadeiras, então as condições de um subcaminho de *path* também serão verdeiras:

Lema 5.

$$\forall(\beta, e, e_f, subpath, path) :$$
$$\exists p : \ e = SA(e_f, p)$$
$$\Rightarrow$$
$$EC(e_f, \Pi(e, path), \beta) \wedge$$
$$suffix(subpath, path)$$
$$\Rightarrow$$
$$EC(e_f, \Pi(e, subpath), \beta)$$
$$Onde\ suffix(b, ab) := \exists k : \ k + +b = ab$$

Esse lema está em pvs0_props conforme a figura 1.1

Demonstração. Sabe-se que o conjunto dos elementos da lista de condições de $\Pi(e, subpath)$ é subconjunto dos elementos da lista de condições de $\Pi(e, path)$, pois *subpath* é sufixo de *path*. Logo se a avaliação de todos os elementos de $\Pi(e, path)$ é verdadeira, então $\Pi(e, subpath)$ também será. □

Nota-se que o lema acima é consequência direta de dois fatos:

1. se *subpath* é sufixo de *path*, então a lista $\Pi(e, path)$ contém todos os elementos da lista $\Pi(e, subpath)$.
2. se lista L tiver apenas condições avaliadas como verdadeiras então uma lista l contém alguns ou todos os elementos de L terá condições verdadeiras.

Para demonstrar o segundo fato prova-se que sempre que se toma uma condição em particular da lista L para formar uma lista unitária e ela é avaliada verdadeira, então a conjunção de todos das condições de L é verdadeira.

O primeiro fato é consequência de $\Pi(e, subpath)$ ser sufixo de $\Pi(e, path)$ e dos sufixos formarem uma relação de subconjunto entre os conjuntos de elementos de listas.

A lista $\Pi(e, subpath)$ ser sufixo de $\Pi(e, path)$ está enunciado no lema abaixo.

Lema 6.

$$\forall(e, subpath, path) :$$
$$suffix(subpath, path) \Rightarrow suffix(\Pi(e, subpath), \Pi(e, path))$$
$$Onde\ suffix(b, ab) := \exists k :\ k + +b = ab$$

O lema acima foi especificado da seguinte forma:

```
suffix_path_cnds_suffix :  LEMMA
FORALL(expr:  Expr)(path,path_ext:  (valid_path(expr))):
suffix?(path, path_ext) IMPLIES
suffix?(path_conditions(expr, path), path_conditions(expr, path_ext))
```

Acima, se há um caminho que é sufixo de outro, então a lista das condições de caminho também será sufixo.

Ele está no arquivo pvs0_props conforme a figura 1.1

Demonstração. Em PVS, suffix foi implementado como uma função, mas a prova se simplifica se usar a propriedade da correção (que foi demonstrada pelo autor e está na biblioteca *structures* em PVS), que está explicito em:

$$\text{Onde suffix}(b, ab) := \exists k :\ k + +b = ab$$

Após esse passo, a prova segue em uma simples indução na soma dos tamanhos da listas *path* e *subpath*, ou seja, assume-se que o lema vale para listas cuja a soma de tamanhos é menor que a soma dos tamanhos das listas *path* e *subpath* para provar que vale também para *path* e *subpath*. □

O lemma abaixo diz que quando há TCC terminação há terminação semântica.

Lema 7.

$$\forall e_f : \mathcal{E}xpr : T_\zeta(e_f) \Rightarrow T_\varepsilon(e_f)$$

Esse lema está no arquivo pvs0_termination conforme a figura 1.1.

Demonstração. Suponha que para alguma expressão e_f vale $T_\zeta(e_f)$, ou seja, existe uma relação bem fundada entre os valores mapeados do argumento real e o argumento da chamada recursiva imediata, para qualquer argumento de e_f dado por um valor $\beta : \mathcal{V}al$. Além disso, a função possui um número limitado de chamadas imediatas. Assim com esses valores dos argumentos forma-se vértices e uma chamada imediata forma uma aresta. Nesse caso temos uma árvore cuja a raiz começa no argumento do primeiro vértice e as folhas são o último chamado recursivo. Portanto, tem-se uma árvore com um número de nós finito e com altura finita, pois um caminho da raiz até a folha contém valores ordenados por uma relação bem fundada. Assim, a altura da árvore n é o número máximo de níveis recursivos a ser utilizado. Logo, $\chi(n, e_f, e_f, \beta)$ é verdadeiro. Portanto, $\forall \beta\ \exists n : \chi(n, e_f, e_f, \beta)$ que, por definição, é $T_\chi(e_f)$. Logo

$$\forall e_f : T_\zeta(e_f) \Rightarrow T_\chi(e_f)$$

. Como foi assumido que $T_\chi(e_f) = T_\varepsilon(e_f)$, então:

$$\forall e_f : T_\zeta(e_f) \Rightarrow T_\varepsilon(e_f)$$

□

Na formalização dessa demonstração, o fato de que só haver a possibilidade de uma árvore infinita não foi usado de maneira direta, pois iria requerir o uso de estruturas de dados como árvores o que aumentaria a complexidade das demonstrações. Para representar uma aresta, usa-se a relação bem fundada, e para um caminho entre um nó e outro nó descendente, usa-se essa relação e uma distância para criar outra relação de caminho entre um nó e seu descendente. $\sqsupset$: $(\mathbb{N} - \{0\}) \to (\mathcal{V}al \times \mathcal{V}al) \to \texttt{bool}$

$$\begin{aligned} \sqsupset (n)(M, m) \quad := \quad & \\ & (n = 1 \wedge m \prec M) \vee \\ & (n > 1 \wedge \exists i : (i \overset{e_f}{\prec} M \wedge \ \sqsupset (n-1)(i, m))) \end{aligned}$$

Acima, a relação $\sqsupset (n)(M, m)$ indica que a relação $\overset{e_f}{\prec}$ separa os valores M e m por uma distância n confome pode ser visualizado abaixo:

$$\overbrace{m \overset{e_f}{\prec} \cdots \overset{e_f}{\prec} M}^{\text{distância de } n\text{'s } \overset{e_f}{\prec}}$$

A relação $a \overset{e_f}{\prec} b$ significa que na avaliação da função descrita pela expressão e_f com argumento b há um chamado da mesma função com o argumento a e $a \prec b$.

Logo a altura de uma árvore onde as arestas são dadas pela relação $\overset{e_f}{\prec}$ é dado pela função $\Omega : \mathcal{V}al \to (\mathbb{N} - \{0\})$:

$$\Omega(a) := min\{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$$

Ou seja, tem-se um conjunto $\{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$, que é o conjunto de todos os naturais n maiores que 0 tais que não há um elemento de $\mathcal{V}al$ a uma distância n de a pela relação $\overset{e_f}{\prec}$. Dentre esses valores, toma-se o menor deles. Essa será a altura da árvore mais um, porque :

- A altura de uma árvore h é a distância entre a raíz a e uma folha qualquer, ou seja, $h \notin \{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$.
- Todo natural não nulo d menor que h serve como distância entre a e um descendente qualquer. Assim, $d \notin \{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$.
- Se $h + 1 \notin \{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$, $h + 1$ seria a altura da árvore e não h, já que a altura é máxima.
- Portanto, $h + 1 \in \{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$.
- Na verdade, a partir de h+1, qualquer natural não pertence a $\{n : (\mathbb{N}-\{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$.
- Logo, $h + 1$ é o menor valor que pertence à $\{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\}$ e, portanto, $min\{n : (\mathbb{N} - \{0\}) \mid \neg\exists\, b :\sqsupset (n)(a, b)\} = \Omega(a) = h + 1$

Apesar de mostrar-se acima que uma árvore com raiz a possui altura $\Omega(a) - 1$, isto não foi formalizado, pois o uso de um grafo dirigido como árvore aumentaria a complexidade, pois iria requerer trabalhar com mais uma estrutura de dados e formalizar propriedades referentes a ela.

Mas no momento em que se percebe de que Ω retorna um valor maior que a altura da árvore pode-se formular o seguinte lema, que diz que, partindo do princípio que e_f é TCC terminante, o limite mímimo do número níveis de chamadas recursivas (expresso pela

função $\mu(e)(v))$ é menor que a altura mais um, quando avaliado o valor v na expressão e_f.

Lema 8.

$$\forall(e_f, v) :$$
$$T_\zeta(e_f) \Rightarrow \mu(e_f)(v) \leq \Omega(v)$$
$$Onde\ \mu(e_f)(v) := min\{m|\chi(m, e_f, e_f, v) \neq \diamondsuit\}$$

Esse lema está no arquivo pvs0_termination conforme 1.1.

Ele foi especificado assim:

```
mu_le_omega:  LEMMA
FORALL((pvs0|pvs0_tcc_termination_pred(pvs0)(wfm)), v:Val):
LET n = Omega[Val,lt_val(pvs0)(wfm)](v) IN
mu(pvs0)(v) <= n
```

Demonstração. A demonstração depende do lema abaixo. Supondo, $T_\zeta(e_f)$ para um e_f qualquer, e assumindo (essa assertiva abaixo foi demonstrada em PVS):

$$\forall\ e_f :$$
$$T_\zeta(e_f) \Rightarrow$$
$$\forall(e, \beta, path)$$
$$e = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\ n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e, e_f, \beta)$$

Supondo $T_\zeta(e_f)$ tem-se:

$$\forall(e, \beta, path)$$
$$e = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\ n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e, e_f, \beta)$$

Acima, escolhe-se e como e_f, β como v, onde v é um valor qualquer e $path$ como []. Logo,

$$\exists\, n$$
$$n \leq \Omega(v) \wedge$$
$$\chi(n, e_f, e_f, v) \neq \diamondsuit \wedge$$
$$\varepsilon(e_f, e_f, v, val)$$
$$\text{Onde } val := \chi(n, e_f, e_f, v)$$

Ou seja, existe um valor n tal que:

$$n \leq \Omega(v) \wedge$$
$$\chi(n, e_f, e_f, v) \neq \diamondsuit$$

Se existe esse valor n, então existirá um mínimo valor m tal que:

$$\chi(m, e_f, e_f, v) \neq \diamondsuit$$

E esse valor m é menor ou igual a n, portanto:

$$m \leq \Omega(v) \wedge$$
$$\chi(m, e_f, e_f, v) \neq \diamondsuit$$

Pela defininição de $\mu(e_f)(v)$, tem-se:

$$\mu(e_f)(v) \leq \Omega(v)$$
$$\text{Onde } \mu(e_f)(v) := min\{m | \chi(m, e, e, v) \neq \diamondsuit\}$$

E como foi suposto $T_\zeta(e_f)$, para todo e_f, v :

$$T_\zeta(e_f) \Rightarrow \mu(e_f)(v) \leq \Omega(v)$$
$$\text{Onde } \mu(e_f)(v) := min\{m | \chi(m, e_f, e_f, v) \neq \diamondsuit\}$$

$$\mu(e)(v) \leq \Omega(v)$$

□

O lema abaixo diz que se tomar uma subexpressão de e_f cujo caminho de condições é valido para um valor β então a árvore de avaliação dessa subexpressão para esse mesmo valor possui tamanho menor ou igual a $\Omega(\beta)$.

Lema 9.

$$\forall\, e_f :$$
$$T_\zeta(e_f) \Rightarrow$$
$$\forall(e, \beta, path)$$
$$e = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e, e_f, \beta, val)$$
$$Onde\ val := \chi(n, e, e_f, \beta)$$

Ele está no arquivo pvs0_termination conforme a figura 1.1

Demonstração. Suponha para um e_f qualquer que vale $T_\zeta(e_f)$. Assim tem que provar:

$$\forall(e, \beta, path)$$
$$e = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e, e_f, \beta)$$

Inicia-se indução na ordem lexicográfica de (β, e) onde:

$$(\beta_1, e_1) \succ (\beta_2, e_2) := \mu(\beta_1) \succ \mu(\beta_2) \vee (\mu(\beta_1) \not\succ \mu(\beta_2) \wedge e_1 > e_2)$$

Acima $e_1 > e_2$ significa que e_2 é subexpressão de e_1.

- Caso e for constante: $e = cnst$.

  Nesse caso, valerá:

  $$\exists\, n$$
  $$n \leq \Omega(\beta) \wedge$$
  $$\chi(n, cnst, e_f, \beta) \neq \diamondsuit \wedge$$
  $$\varepsilon(cnst, e_f, \beta, val)$$
  $$\text{Onde } val := \chi(n, cnst, e_f, \beta)$$

  Basta fazer $n = 1$.

- Caso e for variável: $e = vr$.

Nesse caso, valerá:

$$\begin{gathered}\exists\, n \\ n \leq \Omega(\beta) \wedge \\ \chi(n, vr, e_f, \beta) \neq \diamondsuit \wedge \\ \varepsilon(vr, e_f, \beta, val) \\ \text{Onde } val := \chi(n, vr, e_f, \beta)\end{gathered}$$

Basta fazer $n = 1$.

- Caso e for operador unário: $e = op1(e_1)$

  Por hipótese de indução vale:

$$\begin{gathered}\forall path \\ e_1 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow \\ \exists\, n \\ n \leq \Omega(\beta) \wedge \\ \chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\ \varepsilon(e_1, e_f, \beta, val) \\ \text{Onde } val := \chi(n, e_1, e_f, \beta)\end{gathered}$$

  Supondo $op1(e_1) = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta)$ para um $path$ e β quaisquer é possível concluir:

  $e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta)$

  Assim, na hipótese de indução, fazendo $path := [0] + +path$:

$$\begin{gathered}e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta) \Rightarrow \\ \exists\, n \\ n \leq \Omega(\beta) \wedge \\ \chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\ \varepsilon(e_1, e_f, \beta, val) \\ \text{Onde } val := \chi(n, e_1, e_f, \beta)\end{gathered}$$

  Logo, tem-se:

$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_1, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_1, e_f, \beta)$$

Que implica em:

$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, op1(e_1), e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(op1(e_1), e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, op1(e_1), e_f, \beta)$$

- Caso e for operador binário: $e = op2(e_1, e_2)$

  Por hipótese de indução vale:

$$\forall path$$
$$e_1 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_1, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_1, e_f, \beta)$$

  e também vale:

$$\forall path$$
$$e_2 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_2, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_2, e_f, \beta)$$

  Supondo $op2(e_1, e_2) = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta)$ para um $path$ e um β quaisquer, vale

  $e_1 = SA(e_f, [0] + + path) \wedge EC(e_f, \Pi(e_f, [0] + + path), \beta)$ e vale

$e_2 = SA(e_f, [1] + +path) \wedge EC(e_f, \Pi(e_f, [1] + +path), \beta)$.

Assim, instanciando as hipóteses de indução tem-se:

$$
\begin{gathered}
e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

e também

$$
\begin{gathered}
e_2 = SA(e_f, [1] + +path) \wedge EC(e_f, \Pi(e_f, [1] + +path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_2, e_f, \beta)
\end{gathered}
$$

Portanto as fórmulas podem ser simplificadas em:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

e também em:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_2, e_f, \beta)
\end{gathered}
$$

Elimina-se os quantificadores existenciais:

$$
\begin{gathered}
n_1 \leq \Omega(\beta) \wedge \\
\chi(n_1, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

$$
\begin{gathered}
n_2 \leq \Omega(\beta) \wedge \\
\chi(n_2, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n_2, e_2, e_f, \beta)
\end{gathered}
$$

Assim vale:

$$
\begin{gathered}
max(n_1, n_2) \leq \Omega(\beta) \wedge \\
\chi(max(n_1, n_2), op2(e_1, e_2), e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(op2(e_1, e_2), e_f, \beta, val) \\
\text{Onde } val := \chi(max(n_1, n_2), op2(e_1, e_2), e_f, \beta)
\end{gathered}
$$

Onde $max(n_1, n_2)$ é o maior valor entre n_1 e n_2.

- Caso e for `IF THEN ELSE`: $e = ite(e_1, e_2, e_3)$.

  Por hipótese de indução vale:

$$
\begin{gathered}
\forall path \\
e_1 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

$$
\begin{gathered}
\forall path \\
e_2 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_2, e_f, \beta)
\end{gathered}
$$

$$
\begin{gathered}
\forall path \\
e_3 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_3, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_3, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_3, e_f, \beta)
\end{gathered}
$$

Supondo $ite(e_1, e_2, e_3) = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta)$ para um $path$ e um β quaisquer vale:

$e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta)$

Assim, valerá

$$
\begin{gathered}
e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

e também:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

Eliminando o quantificador existencial, tem-se:

$$
\begin{gathered}
n_1 \leq \Omega(\beta) \wedge \\
\chi(n_1, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Considerando dois casos: $\chi(n_1, e_1, e_f, \beta) = \textbf{TRUE}$ e $\chi(n_1, e_1, e_f, \beta) = \textbf{FALSE}$:

- $\chi(n_1, e_1, e_f, \beta) = \textbf{TRUE}$

Assim, vale $e_2 = SA(e_f, [1] + +path) \wedge EC(e_f, \Pi(e_f, [1] + +path), \beta)$. Logo, também valerá:

$$
\begin{gathered}
e_2 = SA(e_f, [1] + +path) \wedge EC(e_f, \Pi(e_f, [1] + +path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_2, e_f, \beta)
\end{gathered}
$$

Simplificando fica:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_2, e_f, \beta)
\end{gathered}
$$

Eliminando o quantificador existencial:

$$
\begin{gathered}
n_2 \leq \Omega(\beta) \wedge \\
\chi(n_2, e_2, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_2, e_f, \beta, val) \\
\text{Onde } val := \chi(n_2, e_2, e_f, \beta)
\end{gathered}
$$

Logo, vale:

$$
\begin{gathered}
max(n_1, n_2) \leq \Omega(\beta) \wedge \\
\chi(max(n_1, n_2), ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(ite(e_1, e_2, e_3), e_f, \beta, val) \\
\text{Onde } val := \chi(max(n_1, n_2), ite(e_1, e_2, e_3), e_f, \beta)
\end{gathered}
$$

– $\chi(n_1, e_1, e_f, \beta) = \textbf{FALSE}$

Assim, vale $e_3 = SA(e_f, [2] + +path) \wedge EC(e_f, \Pi(e_f, [2] + +path), \beta)$. Logo, também valerá:

$$e_3 = SA(e_f, [2] + +path) \wedge EC(e_f, \Pi(e_f, [2] + +path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_3, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_3, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_3, e_f, \beta)$$

Simplificando fica:

$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_3, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_3, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_3, e_f, \beta)$$

Eliminando o quantificador existencial:

$$n_3 \leq \Omega(\beta) \wedge$$
$$\chi(n_3, e_3, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_3, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n_3, e_3, e_f, \beta)$$

Logo, vale:

$$max(n_1, n_3) \leq \Omega(\beta) \wedge$$
$$\chi(max(n_1, n_3), ite(e_1, e_2, e_3), e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(ite(e_1, e_2, e_3), e_f, \beta, val)$$
$$\text{Onde } val := \chi(max(n_1, n_3), ite(e_1, e_2, e_3), e_f, \beta)$$

- Caso e for a recursão: $e = rec(e_1)$

  Por hipótese de indução tem-se:

$$\forall path$$
$$e_1 = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta) \Rightarrow$$
$$\exists\, n$$
$$n \leq \Omega(\beta) \wedge$$
$$\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(e_1, e_f, \beta, val)$$
$$\text{Onde } val := \chi(n, e_1, e_f, \beta)$$

Supondo $rec(e_1) = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \beta)$ tem-se $e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta)$. Instanciando $path := [0] + +path$ na fórmula anterior, tem-se:

$$
\begin{gathered}
e_1 = SA(e_f, [0] + +path) \wedge EC(e_f, \Pi(e_f, [0] + +path), \beta) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

Eliminando a implicação:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\beta) \wedge \\
\chi(n, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n, e_1, e_f, \beta)
\end{gathered}
$$

Eliminando o quantificador existencial:

$$
\begin{gathered}
n_1 \leq \Omega(\beta) \wedge \\
\chi(n_1, e_1, e_f, \beta) \neq \diamondsuit \wedge \\
\varepsilon(e_1, e_f, \beta, val) \\
\text{Onde } val := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Novamente, por hipótese de indução:

$$
\begin{gathered}
\forall path \\
e_f = SA(e_f, path) \wedge EC(e_f, \Pi(e_f, path), \gamma) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\gamma) \wedge \\
\chi(n, e_f, e_f, \gamma) \neq \diamondsuit \wedge \\
\varepsilon(e_f, e_f, \gamma, val) \\
\text{Onde } val := \chi(n, e_f, e_f, \gamma) \\
\gamma := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

No caso da fórmula anterior faz parte da hipótese de indução, pois, $(\beta, e_1) \succ (\gamma, e_f)$, já que $T_\zeta(rec(e_1))$ então $\beta \succ \gamma$.

Instanciando $path := []$ tem-se:

$$
\begin{gathered}
e_f = SA(e_f, []) \wedge EC(e_f, \Pi(e_f, []), \gamma) \Rightarrow \\
\exists\, n \\
n \leq \Omega(\gamma) \wedge \\
\chi(n, e_f, e_f, \gamma) \neq \diamondsuit \wedge \\
\varepsilon(e_f, e_f, \gamma, val) \\
\text{Onde } val := \chi(n, e_f, e_f, \gamma) \\
\gamma := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Simplificando a implicação:

$$
\begin{gathered}
\exists\, n \\
n \leq \Omega(\gamma) \wedge \\
\chi(n, e_f, e_f, \gamma) \neq \diamondsuit \wedge \\
\varepsilon(e_f, e_f, \gamma, val) \\
\text{Onde } val := \chi(n, e_f, e_f, \gamma) \\
\gamma := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Eliminando o quantificador existencial:

$$
\begin{gathered}
n_2 \leq \Omega(\gamma) \wedge \\
\chi(n_2, e_f, e_f, \gamma) \neq \diamondsuit \wedge \\
\varepsilon(e_f, e_f, \gamma, val) \\
\text{Onde } val := \chi(n_2, e_f, e_f, \gamma) \\
\gamma := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Como $\Omega(\gamma) < \Omega(\beta)$ então:

$$
\begin{gathered}
n_2 \leq \Omega(\beta) \wedge \\
\chi(n_2, e_f, e_f, \gamma) \neq \diamondsuit \wedge \\
\varepsilon(e_f, e_f, \gamma, val) \\
\text{Onde } val := \chi(n_2, e_f, e_f, \gamma) \\
\gamma := \chi(n_1, e_1, e_f, \beta)
\end{gathered}
$$

Portanto:

$$max(n_1, n_2) \leq \Omega(\beta) \wedge$$
$$\chi(max(n_1, n_2), rec(e_1), e_f, \beta) \neq \diamondsuit \wedge$$
$$\varepsilon(rec(e_1), e_f, \beta, val)$$
$$\text{Onde } val := \chi(max(n_1, n_2), rec(e_1), e_f, \beta)$$

□

Logo, foram formalizados 269 TCC‘s e lemas no total para a biblioteca PVS0.

Capítulo 5

Formalização da indecidibilidade do problema da parada e Turing Completude de PVS0

Além das formalizações de indecidibilidade do Problema da Parada e Turing Completude de PVS0, contribuição principal desta dissertação [18], serão apresentadas aplicações da teoria PVS0 para determinação da terminação de funções especificadas na linguagem PVS.

5.1 Indecidibilidade do Problema da Parada para PVS0

Usando a equivalência entre as noções de terminação apresentadas no Capítulo 4, foi formalizada a indecidibilidade do problema da parada. Para que o resultado seja significante, é preciso também garantir que a linguagem PVS0 é tão expressiva quanto as linguagens de programação, ou seja, que ela é Turing Completa.

A indecidibilidade do problema da parada está especificada no seguinte lema.

Lema 10.

$$\exists\,(\mathit{Val2Expr} : \mathcal{V}al \to \mathcal{E}xpr) : \forall(\mathit{expr} : \mathcal{E}xpr) :\ \exists(\nu : \mathcal{V}al) :\ \mathit{Val2Expr}(\nu) = \mathit{expr}$$
$$\Rightarrow \qquad \neg\exists\,(\mathit{halt} : \mathcal{E}xpr \to \texttt{bool}) : \forall\, e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$$

Esse lema está no arquivo pvs0_undecidability conforme a figura 1.1. Para essa formalização foi assumido que existe um mapeamento sobrejetivo entre os valores de $\mathcal{V}al$ e as expressões PVS0. Essa é uma premissa natural que valerá sempre que o tipo não interpretado $\mathcal{V}al$ é um conjunto infinitamente enumerável.

Demonstração. Suponha $\exists$ ($\mathit{Val2Expr} : \mathcal{V}al \rightarrow \mathcal{E}xpr$) : $\forall(\mathit{expr} : \mathcal{E}xpr) : \exists(\nu : \mathcal{V}al) : \mathit{to\_}\mathcal{E}\mathit{xpr}(\nu) = \mathit{expr}$. Este mapeamento é chamado *Val2Expr*. Para obter uma contradição, suponha também:

$$\exists\,(\mathit{halt} : \mathcal{E}\mathit{xpr} \rightarrow \texttt{bool}) : \forall\, e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$$

Logo, existe a função *halt* com essa propriedade. A expresssão e_f será escolhida como $\mathit{ite}(H(vr), \mathit{rec}(vr), vr)$, que para abreviar será chamada $m := \mathit{ite}(H(vr), \mathit{rec}(vr), vr)$.

Portanto, o resultado é o seguinte: $\mathit{halt}(m) \Leftrightarrow T_\varepsilon(m)$.

Ambos os casos $\mathit{halt}(m)$ e $\neg\mathit{halt}(m)$ são analisados.

Caso $\mathit{halt}(m)$.

Se $\forall\, e_f : \mathit{halt}(e_f) \Leftrightarrow T_\varepsilon(e_f)$ vale para uma expressão e_f, também vale para uma expressão $\mathit{ite}(H(vr), \mathit{rec}(vr), vr)$, onde $H^\Im := \mathit{halt} \circ \mathit{Val2Expr}$ ($\circ$ é a composição de funções). Neste caso, $m = \mathit{ite}(H(vr), \mathit{rec}(vr), vr)$.

Isso resulta em $(\mathit{halt}(m) \Rightarrow T_\varepsilon(m)) \wedge (T_\varepsilon(m) \Rightarrow \mathit{halt}(m))$. Pela definição de T_ε, logo, $\forall\beta, \exists\nu : \varepsilon(m, m, \beta, \nu)$.

Escolhendo $\beta := M_0$, onde $\mathit{Val2Expr}(M_0) = m$ (o valor M_0 existe pela sobrejeção da função *Val2Expr*), $\varepsilon(m, m, \beta, \nu)$ entra em processamento infinito para cada valor ν e, portanto, o valor ν não existe, o que é uma contradição.

O assistente de prova PVS não permite esse tipo de demonstração por causa do processamento infinito de ε. Neste caso , foi aplicado o seguinte resultado :

$$T_\varepsilon(m) \Rightarrow \forall\beta\ \chi(\mu(m)(\beta), m, m, \beta) \neq \diamondsuit$$

Esse resultado é provado pela própria definição de μ:

Se vale $T_\varepsilon(m)$, vale $T_\chi(m)$. Assim, para todo β o conjunto $\{k \mid \chi(k, m, m, \beta) \neq \diamondsuit\}$ é não vazio. Logo, esse conjunto possui um menor elemento que é $\mu(m)(\beta)$, por definição. Logo, vale $\forall\beta\ \chi(\mu(m)(\beta), m, m, \beta) \neq \diamondsuit$.

Portanto, por consequência de $T_\varepsilon(m)$, vale $\forall\beta\ \chi(\mu(m)(\beta), m, m, \beta) \neq \diamondsuit$. Escolhendo $\beta := M_0$ tem-se: $\chi(\mu(m)(M_0), m, m, M_0) \neq \diamondsuit$. Assim, se expandir e simpilficar χ tem-se $\chi(\mu(m)(M_0) - 1, m, m, M_0) \neq \diamondsuit$. Assim $\mu(m)(M_0) - 1$ seria o valor mínimo o que contradiz a minimalidade de $\mu(m)(M_0)$.

Caso $\neg\mathit{halt}(m)$.

Tem-se que $\neg T_\varepsilon(m)$ vale. Pela expanção da definição: $\neg\forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon(m, m, \beta, \nu)$, que é equivalente a: $\exists\ \beta,\ \forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(m, m, \beta, \nu)$. Logo, existe β, tal que: $\forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(m, m, \beta, \nu)$. Expandindo a definição de ε e considerando $\neg\mathit{halt}(m)$: $\forall\ \nu \in \mathcal{V}al :\ \neg\varepsilon(vr, m, \beta, \nu)$.

Para concluir, um valor ν diferente de β é escolhido, o que gera uma contradição. □

Nota-se que a mesma técnica de diagonalização e a godelização utilizadas no Capítulo 2 para Máquinas de Turing e Máquinas com Registros são usadas nessa formalização em PVS0.

Essa formalização possui 77 linhas de comandos de prova.

5.2 Formalização de Turing-Completude de PVS0

Desde que a indecidibilidade do problema da parada para modelos computacionais restritos e não expressivos é irrelevante, também é necessário provar que PVS0 é uma linguagem Turing-Completa. Para formalizar Turing completude foi provado que PVS0 codifica as funções recursivas básicas: zero, sucessor e projeções tão bem quanto adição, multiplicação e a função *chi* (que é, $chi(x, y) = 1$ if $x \leq y$ else 0), e que ela é fechada sobre as operações de composição e minimização.

A Turing Completude de PVS0 está expressa no lema abaixo, que diz que se existe uma função sobrejetiva que codifica cada expressão PVS0 como um valor em $\mathcal{V}al$ e, existe um codificação de $\mathcal{V}al$ usando números naturais através de uma função bijetiva, então a languagem PVS0 computa todas as funções parciais recursivas.

Lema 11 (Turing Completude de PVS0)**.**

$$\begin{aligned} \exists\,(\mathit{Val2Expr} : \mathcal{V}al \to \mathcal{E}xpr) : \forall(\mathit{expr} : \mathcal{E}xpr) :\ & \exists(\nu : \mathcal{V}al) :\ \mathit{Val2Expr}(\nu) = \mathit{expr} \\ & \wedge \quad \exists\,(\mathit{Val2Nat} : \mathcal{V}al \to \mathbb{N}) :\ \mathit{Val2Nat\ is\ bijective} \\ \Rightarrow \qquad & \texttt{PVS0}\ \mathit{é\ Turing\text{-}Completa} \end{aligned}$$

Esse lema está localizado no arquivo pvs0_comp conforme a figura 1.1.

Demonstração. Se existe um função bijetiva $\mathit{Val2Nat} : \mathcal{V}al \to \mathbb{N}$, então é possível codificar sucessor, adição e multiplicação. Considerando *Val2Nat* uma bijeção entre $\mathcal{V}al$ e $\mathbb{N}$, e *Nat2Val* a função inversa. Os símbolos S, P e M implementam sucessor, adição e multiplicação respectivamente:

- $S^{\mathfrak{I}}(v) = \mathit{Nat2Val}(\,\mathit{Val2Nat}(v) + 1)$
- $P^{\mathfrak{I}}(v_1, v_2) = \mathit{Nat2Val}(\,\mathit{Val2Nat}(v_1) + \mathit{Val2Nat}(v_2))$
- $M^{\mathfrak{I}}(v_1, v_2) = \mathit{Nat2Val}(\,\mathit{Val2Nat}(v_1) \times \mathit{Val2Nat}(v_2))$

Existe uma função bijetiva entre $\mathbb{N}$ e listas não vazias de $\mathbb{N}$. Logo, existe um função bijetiva entre $\mathcal{V}al$ e listas não vazias de $\mathbb{N}$. Chamando essa função de *to_List* e considerando $rem(a, b)$ como o resto da divisão entre a e b; $nth(l, i)$ como o i^{th} elemento da lista

l (iniciando de 0 até o tamanho de l menos um) em uma lista l; e $length$ como o número de elementos em uma lista, a projeção P_i é implementeda como:

$$P_i^{\mathfrak{I}}(a,b) = nth(to\_List(a), rem(length(to\_List(a)),\ Val2Nat(b)))$$

A função *chi* é construida como:

$$C_{hi}^{\mathfrak{I}}(a,b) = if\ \ Val2Nat(a) \leq Val2Nat(b)\ then\ Nat2Val(1)\ else\ Nat2Val(0)$$

Para a composição, considere $Val2Expr : \mathcal{V}al \rightarrow \mathcal{E}xpr$ como uma função sobrejetiva e a função $Expr2Val : \mathcal{E}xpr \rightarrow \mathcal{V}al$ dada como:

$$Expr2Val(e) = choose(\{v : \mathcal{V}al |\, Val2Expr(v) = e\})$$

onde *choose* é uma função que escolhe um valor de um conjunto não vazio. Considere também as funções:

- $O^{\mathfrak{I}}(a,b) = choose(\{v : \mathcal{V}al | if\ \exists\ t : \varepsilon(\ Val2Expr(a),\ Val2Expr(a), b, t)\ then$
  $\varepsilon(\ Val2Expr(a),\ Val2Expr(a), b, v)\ else\ v = Nat2Val(0)\})$
- $O_{aux}^{\mathfrak{I}}(a,b) = choose(\{v : \mathcal{V}al | if\ \exists\ t : \varepsilon(\ Val2Expr(a),\ Val2Expr(a), b, t)\ then$
  $v = Nat2Val(1)\ else\ v = Nat2Val(0)\})$

Logo, a composição é construida como:

$$\begin{aligned} Comp(f,g) \ :=\ & ite(O_{aux}(Expr2Val(g), vr), \\ & \quad ite(O_{aux}(Expr2Val(f), O(Expr2Val(g), vr)), \\ & \quad\ O(Expr2Val(f), O(Expr2Val(g), vr)), \\ & \quad\ rec(vr)) \\ & \quad rec(vr)) \end{aligned}$$

A instrução de ramificação *ite* considera $Nat2Val(0)$ como falso e qualquer valor diferente como true. Como dito na introdução, a linguagem PVS0 não apresenta mecanismos naturais para compor, tais como cálculo lambda, cálculo combinatório, Máquinas de Turing e Máquinas com Registros. Em Máquinas de Turing, por exemplo, a composição das máquinas $M_1 = \langle Q_1, q_i, q, \Gamma, \Sigma, \delta\rangle$ e $M_2 = \langle Q_2, q_i', q', \Gamma, \Sigma, \delta_2\rangle$ considerando Q_1 e Q_2 conjuntos de estados disjuntos, seria Executar M_1, no estado de aceitação de q, mudar para um estado diferente de todos os outros r, acrescentar estados para voltar ao começo da fita, e executar M_2. Essa facilidade não está disponível na linguagem PVS0; assim, a construção da composição necessitou da godelização, porque qualquer outra forma de se tentar compor funções especificadas nesta linguagem de forma general falha. Note que

utilizar operadores binários não é possível: dado um operador binário C, este toma dois valores em $\mathcal{V}al$ e retorna outro valor em $\mathcal{V}al$. Assim, se tentar compor as expressões PVS0 f e g por $C(f, g)$ não vai funcionar, pois qualquer interpretação de símbolos de operadores binários não trabalha diretamente transformando expressões PVS0 em outras expressões. O problema é que a avaliação semântica de chamados recursivos não poderá discriminar, quando a computação de g finaliza e quando a de f deve ser ativada; assim, uma expressão do tipo $O(f, g)$ não comporta como composição. Tentativas similares utilizando a instrução de ramificação apresentam o mesmo problema, quando se busca uma construção da composição. Em modelos computacionais como cálculo lambda, modelos funcionais e imperativos a composição aparece naturalmente via a operaç ão de aplicação funcional básica da linguagem.

O predicado auxiliar $min : \mathcal{E}xpr \times \mathcal{V}al \times \mathcal{V}al \rightarrow \texttt{bool}$ especifica minimização:

$$\begin{array}{rcl} min(e,t,v) & := & \textbf{IF } \varepsilon(e,e,t,Nat2Val(0)) \textbf{ THEN } t = v \\ & & \textbf{ELSE } min(e, Nat2Val(\,Val2Nat(t)+1), v) \end{array}$$

Os símbolos M_{in} e M_{aux} representam minimização:

- $M^{\mathfrak{I}}_{in}(a,b) = choose(\{v : \mathcal{V}al | if\ \exists\ t\ min(\,Val2Expr(a), b, t)\ then$
  $min(\,Val2Expr(a), b, v)\ else\ v = Nat2Val(0)\})$
- $M^{\mathfrak{I}}_{aux}(a,b) = choose(\{v : \mathcal{V}al | if\ \exists\ t\ min(\,Val2Expr(a), b, t)\ then$
  $v = Nat2Val(1)\ else\ v = Nat2Val(0)\})$

Finalmente, minimização é representado como:

$$MIN(f) \ := \ ite(M_{aux}(Expr2Val(f), vr), M_{in}(Expr2Val(f), vr), rec(vr))$$

Para demonstrar que MIN minimiza, é necessário demonstrar que o predicado min vale para o valor v tal que a avaliação da expressão e com entrada v resulta em $Nat2Val(0)$: $\forall(e,t,v) : min(e,t,v) \Rightarrow \varepsilon(e,e,v,Nat2Val(0))$.

Isso procede por indução na estrutura de min. Isso significa que se a propriedade vale para a recursão deve-se mostrar que ela vale para o procedimento min.

- Caso $\varepsilon(e,e,t,Nat2Val(0))$ então $t = v$. Logo, $\varepsilon(e,e,v,Nat2Val(0))$.
- Caso $\neg\varepsilon(e,e,t,Nat2Val(0))$ então $min(e, Nat2Val(\,Val2Nat(t)+1), v)$. Por hipótese de indução, $\varepsilon(e,e,v,Nat2Val(0))$.

Para formalizar que o predicado *min* realmente minimiza, requer provar que a seguinte propriedade, que sinaliza que *min* vale para um mínimo valor v:

$$\forall(e,t,v) : min(e,t,v) \Rightarrow \forall h : Val2Nat(t) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e,e,h,Nat2Val(0))$$

A prova é também por indução na estrutura de *min*.

- Caso $\varepsilon(e,e,t,Nat2Val(0))$ então $t = v$. Logo $\forall h : Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e,e,h,Nat2Val(0))$. A assertiva $Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v)$ é falsa, então é verdade que $\forall h : Val2Nat(v) \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e,e,h,Nat2Val(0))$.
- Caso $\neg\varepsilon(e,e,t,Nat2Val(0))$ então $min(e,Nat2Val(Val2Nat(t)+1),v)$. Por hipótese de indução,

  $\forall h : Val2Nat(t)+1 \leq Val2Nat(h) < Val2Nat(v) \Rightarrow \neg\varepsilon(e,e,h,Nat2Val(0))$. Se $t = h$, então $\neg\varepsilon(e,e,h,Nat2Val(0))$ vale. Então, $\forall h : Val2Nat(t) \leq Val2Nat(h) < to\_Nat(v) \Rightarrow \neg\varepsilon(e,e,h,Nat2Val(0))$.

□

Para formalização da composição foi necesssária 463 linhas de comandos de prova.

Já para minimização, a formalização precisou de 2 lemas auxiliares e foram necessárias 110 linhas de comandos de prova prova.

5.3 Aplicações da teoria PVS0

Como exemplo de uso da teoria PVS0 foi demonstrada a terminação das funções de Ackermann e gcd. A função de Ackermann foi especificada da seguinte maneira:

```
IMPORTING ack_pvs0
    ack(m,n:nat) :  RECURSIVE {a :  nat | a = ack_pvs0(m,n)} =
        IF m = 0 THEN n+1
        ELSIF n = 0 THEN ack(m-1,1)
        ELSE ack(m-1,ack(m,n-1))
    ENDIF
MEASURE ack_wfm BY R
```

Na primeira linha em `IMPORTING ack_pvs0` indica que houve a importação de propriedades e funções do arquivo ack_pvs0. Na segunda linha, `{a :  nat | a = ack_pvs0(m,n)}` indica que o tipo de retorno da função `ack` é dos elementos da função `ack_pvs0(m,n)`, que foi especificada no arquivo supracitado usando a linguagem PVS0.

Na última linha, `ack_wfm` é uma função que leva uma tupla de naturais a um espaço de medidas dos ordinais e `R` é uma relação de ordem dos números ordinais. Essa função e essa relação estão definidas em ack_pvs0, mas `ack_wfm` é uma função qualquer de tal forma que mantém a propriedade de TCC terminação de `ack_pvs0`.

Para isso é necessário explicitar que existe uma função, mostrando que `ack_pvs0` obedece a propriedade de terminação.

A função `ack_pvs0` está definida segundo a seguinte codificação em PVS0:

```
pvs0_ack :  Def =
def(ite(op1(0,vr),
     op1(2,vr),
     ite(op1(1,vr),
          rec(op1(3,vr)),
          rec(op2(0,vr,rec(op1(4,vr)))))))
```

Lembrando que `ack_pvs0` é uma função de tupla de naturais em naturais que usa `pvs0_ack`, que é a codificação de uma função de Ackermann de tupla naturais em tupla de naturais cujo resultado fica na primeira parte da tupla.

A demonstração de que é terminante envolve ordem dos ordinais. Assim tem-se a função $lex2 : \mathbb{N} \times \mathbb{N} \to \mathcal{O}$, onde $\mathcal{O}$ é o conjunto dos ordinais.

$$lex2(x, y) := x\omega + y$$

O simbolo $<$ e similares, além de usado para denotar ordem dos números reais, será usado para denotar ordem dos ordinais.

No caso acima, tem-a equivalência:

$$lex2(a, b) < lex2(m, n) \equiv a < m \vee (a = m \wedge b < n)$$

Já está provado na biblioteca do prelúdio (biblioteca do PVS com várias propriedades matemáticas já demonstradas) que a ordem dos ordinais é uma relação bem fundada e a equivalência acima.

Assim, `ack_pvs0` termina, pois `pvs0_ack` obedece a propriedade de TCC terminação usando ordinais como espaço de medidas e $lex2$ como função de medida.

No caso de *gcd* a prova segue um caminho diferente do exemplo do capítulo 3 e usa técnicas semelhantes ao exemplo da função de Ackermann. A função *gcd* está especificada no arquivo gcd_pvs:

```
IMPORTING gcd_pvs0
     gcd(m,n:nat) :  RECURSIVE {a:  nat | a = gcd_pvs0(m,n)} =
          IF m = 0 OR n = 0 THEN m + n
```

```
        ELSIF n >= m THEN gcd(m,n-m)
        ELSE gcd(n,m)
        ENDIF
    MEASURE gcd_wfm BY R
```

Na primeira linha do código acima, em `IMPORTING gcd_pvs0` importa o arquivo gcd_pvs0 que contém propriedades e funções usadas acima. Na segunda linha em `{a : nat | a = gcd_pvs0(m,n)}`, especifica-se que a função `gcd` possui como saída `gcd_pvs0(m,n)`, que é o um algoritmo de máximo divsor comum, mas baseado no código abaixo escrito em linguagem `PVS0`:

```
pvs0_gcd : Def =
def(ite(op1(0,vr),
    op1(2,vr),
     ite(op1(1,vr),
        rec(op1(3,vr)),
        rec(op1(4,vr)))))
```

O código acima é o mesmo escrito no capítulo 3, onde os símbolos são representados como números naturais. Ainda nele, na última linha, está escrito `MEASURE gcd_wfm BY R`, que é onde se define as condições de parada da função `gcd`. A relação `R` está definida como a ordem dos ordinais, e `gcd_wfm`, como uma função qualquer de tupla de naturais para o espaço de medida dos ordinais tal que obedece à propriedade de TCC terminação para `pvs0_gcd` com a relação `R`. Mas para isso é preciso provar que `pvs0_gcd` termina. A prova usa a mesma ideia da função de Ackermann, ou seja, usa como função para um espaço de medida bem fundado a função $lex2$.

Capítulo 6

Conclusão e Trabalhos Futuros

Apresentou-se a linguagem PVS0, as definições semântica de terminação e TCC terminação, a formalização de equivalência entre elas, a formalização de indecidibilidade do problema da parada e de Turing-completude de PVS0.

Para as noções semânticas de terminação a formalização da definição T_ε ser equivalente à definição T_χ provem de uma indução estrutural na definição de ε, que é a função que expressa a semântica de avaliação de programas PVS0.

Já a formalização da noção T_ζ, TCC terminação, implicar na definição T_χ é devido a que se os argumentos de chamadas aninhadas de uma expressão recursiva estão mapeados para um espaço de medidas e sempre ocorre que estão relacionados por uma ordem bem fundada, então a altura da árvore de recursão é finita. Já para a formalização de que T_χ implica na definição de T_ζ é devido a que se para todo valor de entrada existe uma altura de árvore de recursão, então use os naturais como espaço de medidas, a altura como função de medida e a ordem dos naturais como ordem bem fundada.

Formalizou-se a indecidibilidade do problema da parada usando o menor valor para avaliação semântica por χ resultar algo diferente de $\diamondsuit$, pois somente usando terminação semântica há uma expansão indefinida de ε. Mas até então não se sabia se era um resusltado de real interesse computacional, pois PVS0 poderia não ser tão expressivo quanto as linguagens de programação.

Por isso formalizou-se que PVS0 é Turing-Completo, mostrando que nessa linguagem tem como se codificar a função zero, sucessor, adição, multiplicação e que PVS0 é fechado para composição e minimização.

Adicionalmente, é importante mencionar possíveis ampliações do trabalho.

- Nota-se que a linguagem PVS0 trabalha com um só tipo, tanto para entrada quanto para saída das expressões. Nisso gera-se a necessidade de extendê-la para trabalhar com vários tipos, permitindo que a expressão recursiva contenha vários argumentos. A linguagem seria chamada PVS1 e teria a seguinte gramática:

$$\mathcal{E}xpr ::= rec(\mathcal{E}xpr, \cdots, \mathcal{E}xpr) | op1(\mathcal{E}xpr) | op2(\mathcal{E}xpr, \mathcal{E}xpr) | ite(\mathcal{E}xpr, \mathcal{E}xpr, \mathcal{E}xpr) | vr_n | cnst$$

Na gramática acima, vr_n seria a n-ésima variável.

A dificuldade seria encontrar uma estrutura de dados para representar os vários tipos para os argumentos da função recursiva e o tipo de saída. A solução seria usar o coproduto dos tipos apresentados e operar com ele. Resolvendo o problema de tipos relacionado a PVS1 a semântica seria dada pela seguinte relação $\varepsilon 1 : \mathcal{E}xpr \times \mathcal{E}xpr \times [\mathcal{V}al] \times \mathcal{V}al \to \texttt{bool}$:

$$\begin{aligned}
\varepsilon 1(e, e_f, \beta, \nu) \quad := \quad & \textbf{CASES } e \textbf{ OF} \\
& cnst: \quad \nu = cnst^{\mathfrak{I}}; \\
& vr_n: \quad \nu = nth(\beta, n); \\
& op1(e_1): \quad \exists \nu_1 \in \mathcal{V}al : \varepsilon 1(e_1, e_f, \beta, \nu_1) \wedge \\
& \qquad \nu = op1^{\mathfrak{I}}(\nu_1); \\
& op2(e_1, e_2): \quad \exists \nu_1, \nu_2 \in \mathcal{V}al : \varepsilon 1(e_1, e_f, \beta, \nu_1) \wedge \varepsilon 1(e_2, e_f, \beta, \nu_2) \wedge \\
& \qquad \nu = op2^{\mathfrak{I}}(\nu_1, \nu_2); \\
& ite(e_1, e_2, e_3): \quad \exists \nu_1 : \varepsilon 1(e_1, e_f, \beta, \nu_1) \wedge \\
& \qquad \textbf{IF } \nu_1 \textbf{ THEN } \varepsilon 1(e_2, e_f, \beta, \nu) \textbf{ ELSE } \varepsilon 1(e_3, e_f, \beta, \nu); \\
& rec(e_1, \cdots, e_s): \quad \exists \beta' \in [\mathcal{V}al] : \varepsilon 1(e_1, e_f, \beta, nth(\beta', 1)) \wedge \cdots \wedge \\
& \qquad \varepsilon 1(e_s, e_f, \beta, nth(\beta', s)) \wedge \\
& \qquad \varepsilon 1(e_f, e_f, \beta', \nu)
\end{aligned}$$

Acima nth recebe uma lista e um número natural i e retorna o i-ésimo elemento dessa lista. A principal diferença entre ε e $\varepsilon 1$ é que a expressão em ε recebe como entrada um único valor e $\varepsilon 1$ recebe como entrada uma lista de valores. A opção de tratar os diferentes tipos de objetos da linguagem como elementos fixos ordenados nas listas de entrada do mecanismo de avaliação, não é muito diferente do que se pode fazer direitamente com PVS0 via uso de tuplas, como foi feito em gcd. Uma das definições de terminação semântica seria:

$$T_{\varepsilon 1}(e_f) := \forall\ \beta,\ \exists\ \nu \in \mathcal{V}al :\ \varepsilon 1(e_f, e_f, \beta, \nu)$$

Outra maneira de definir terminação semântica seria com a função $\chi 1 : \mathbb{N} \times \mathcal{E}xp \times \mathcal{E}xp \times [\mathcal{V}al] \to \mathcal{V}al \cup \{\diamondsuit\}$.

$$
\begin{array}{rcl}
\chi 1(m, e, e_f, \beta) & := & \textbf{IF } m = 0 \textbf{ THEN } \diamondsuit \\
& & \textbf{ELSE } \quad \textbf{CASES } e \textbf{ OF} \\
& & \qquad cnst: \; cnst^{\mathfrak{I}}; \\
& & \qquad vr_n: \; nth(\beta, n); \\
& & \qquad op1(e_1): \; \textbf{IF } \chi 1(m, e_1, e_f, \beta) \neq \diamondsuit \\
& & \qquad\quad \textbf{THEN } op1^{\mathfrak{I}}(\chi 1(m, e_1, e_f, \beta)) \\
& & \qquad\quad \textbf{ELSE } \diamondsuit; \\
& & \qquad op2(e_1, e_2): \; \textbf{IF } \chi 1(m, e_1, e_f, \beta) \neq \diamondsuit \wedge \chi 1(m, e_2, e_f, \beta) \neq \diamondsuit \\
& & \qquad\quad \textbf{THEN } op2^{\mathfrak{I}}(\chi 1(m, e_1, e_f, \beta), \chi 1(m, e_2, e_f, \beta)) \\
& & \qquad\quad \textbf{ELSE } \diamondsuit; \\
& & \qquad ite(e_1, e_2, e_3): \; \textbf{IF } \chi 1(m, e_1, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
& & \qquad\quad \textbf{IF } \chi 1(m, e_1, e_f, \beta) \textbf{ THEN } \chi 1(m, e_2, e_f, \beta) \\
& & \qquad\quad \textbf{ELSE } \chi 1(m, e_3, e_f, \beta); \\
& & \qquad\quad \textbf{ELSE } \diamondsuit; \\
& & \qquad rec(e_1, \cdots, e_s): \; \textbf{IF } \forall n \leq s \; \chi 1(m, e_n, e_f, \beta) \neq \diamondsuit \textbf{ THEN} \\
& & \qquad\quad \chi 1(m-1, e_f, e_f, \beta'), \\
& & \qquad\quad \text{onde } \beta' = [\chi 1(m, e_1, e_f, \beta), \cdots, \chi 1(m, e_s, e_f, \beta)] \\
& & \qquad\quad \textbf{ELSE } \diamondsuit
\end{array}
$$

A nova definição de terminação semântica seria:

$$T_{\chi 1}(e_f) := \forall\, \beta,\ \exists\, m \in \mathbb{N}: \ \textbf{Seja } \nu := \chi 1(m, e_f, e_f, \beta) \textbf{ em } \nu \neq \diamondsuit \ \wedge \ \varepsilon 1(e_f, e_f, \beta, \nu)$$

TCC terminação para PVS1 seria:

$$
\begin{array}{rcl}
T_{\zeta 1}(e_f) & := & \exists\, \mu,\ \forall\, (\beta,\ cc : Valid\_cc(e_f),\ \nu): \\
& & \varepsilon 1(nth(get\_arg(rec\_expr(cc)), 1), e_f, \beta, nth(\nu, 1)) \wedge \cdots \wedge \\
& & \varepsilon 1(nth(get\_arg(rec\_expr(cc)), s), e_f, \beta, nth(\nu, s)) \wedge \\
& & EC(e_f, cnds(cc), \beta) \Longrightarrow \mu(\nu) \prec \mu(\beta)
\end{array}
$$

No caso acima, *get_arg* retorna a uma lista com expressões que são argumentos de chamados recursivos e μ mapeia uma lista de valores em um espaço de medida que pode ser comparado com a relação bem fundada $\prec$. O teorema a ser provado é o de equivalência entre as noções de terminação definidas acima que seria:

$$\forall e_f : T_{\varepsilon 1}(e_f) \equiv T_{\chi 1}(e_f) \equiv T_{\zeta 1}(e_f)$$

- Deseja-se saber como traduzir de maneira conservativa PVS1 para PVS0 e relacionar as noções de terminação entre cada uma das linguagens como equivalentes. A principal dificuldade dessa proposta seria como traduzir o caso recursivo, pois PVS1 pode conter várias expressões para serem traduzidas para uma única expressão PVS0. Chamando esse tradutor de γ os teoremas a serem provados seriam:

$$\forall e_f : T_{\varepsilon 1}(e_f) \equiv T_{\varepsilon}(\gamma(e_f))$$

$$\forall e_f : T_{\chi 1}(e_f) \equiv T_{\chi}(\gamma(e_f))$$

$$\forall e_f : T_{\zeta 1}(e_f) \equiv T_{\zeta}(\gamma(e_f))$$

  A função de tradução seria implementada da seguinte forma:

$$\begin{array}{l} \gamma(e) \ := \\ \qquad \textbf{CASES } e \textbf{ OF} \\ \qquad\qquad \begin{array}{rl} cnst: & cnst; \\ vr_n: & \pi_n(vr); \\ op1(e_1): & op1(\gamma(e_1)); \\ op2(e_1, e_2): & op2(\gamma(e_1), \gamma(e_2)); \\ ite(e_1, e_2, e_3): & ite(\gamma(e_1), \gamma(e_2), \gamma(e_3)); \\ rec(e_1, \cdots, e_s): & rec(\gamma(e_1) + + \cdots + + \gamma(e_s), []) \end{array} \end{array}$$

  Onde π_n acessa o n-ésimo elemento de uma lista e a coloca como uma lista unitária. Se PVS1 trabalha com o tipo T, PVS0 trabalha com o tipo $[T]$.

- A teoria PVS0 pode ser usada para formalizar propriedades de complexidade. Basta modificar a função χ para contar o número de chamadas recursivas na árvore de recursão para expressões terminantes ou usar a função $\mu(\nu)(e_f)$. Nesse caso pode-se demonstrar teoremas clássicos de complexidade como lema da aceleração e hierarquia de tempo.

Referências

[1] *Biblioteca PVS para Calling Context Graphs e Matrix Weighted Graphs, NASA LaRC PVS library.* `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`. Acessado em: 2017-02-19. 6, 7

[2] *René Thiemann and Christian Sternagel. IsaFoR - Isabelle Formalization of Rewriting.* `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`. Acessado em: 2017-03-13. 7

[3] Arts, Thomas e Jürgen Giesl: *Termination of term rewriting using dependency pairs.* Theor. Comput. Sci., 236(1-2):133–178, 2000. `http://dx.doi.org/10.1016/S0304-3975(99)00207-8`. 7

[4] Avelar, Andréia Borges: *Formalização da automação da terminação através de grafos com matrizes de medida.* Tese de Doutoramento, Universidade de Brasília, Departamento de Matemática, 2015. 2, 6, 7, 31, 40

[5] Blanqui, Frédéric e Adam Koprowski: *CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates.* Mathematical Structures in Computer Science, 21(4):827–859, 2011. `http://dx.doi.org/10.1017/S0960129511000120`. 7

[6] Codish, Michael, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs e Jürgen Giesl: *SAT-based termination analysis using monotonicity constraints over the integers.* TPLP, 11(4-5):503–520, 2011. `http://dx.doi.org/10.1017/S1471068411000147`. 7

[7] Cook, Byron, Andreas Podelski e Andrey Rybalchenko: *Proving thread termination.* Em *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, páginas 320–330, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-633-2. `http://doi.acm.org/10.1145/1250734.1250771`. 1

[8] Cook, Byron, Andreas Podelski e Andrey Rybalchenko: *Proving program termination.* Commun. ACM, 54(5):88–98, maio 2011, ISSN 0001-0782. `http://doi.acm.org/10.1145/1941487.1941509`. 1

[9] Cook, Byron, Abigail See e Florian Zuleger: *Ramsey vs. lexicographic termination proving.* Em *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 47–61. Springer, 2013. 8

[10] Falke, Stephan, Deepak Kapur e Carsten Sinz: *Termination analysis of C programs using compiler intermediate languages*. Em Schmidt-Schauß, Manfred (editor): *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 de *LIPIcs*, páginas 41–50. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, ISBN 978-3-939897-30-9. http://dx.doi.org/10.4230/LIPIcs.RTA.2011.41. 7

[11] Falke, Stephan, Deepak Kapur e Carsten Sinz: *Termination analysis of imperative programs using bitvector arithmetic*. Em Joshi, Rajeev, Peter Müller e Andreas Podelski (editores): *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 de *Lecture Notes in Computer Science*, páginas 261–277. Springer, 2012, ISBN 978-3-642-27704-7. http://dx.doi.org/10.1007/978-3-642-27705-4_21. 7

[12] Giesl, Jürgen, René Thiemann e Peter Schneider-Kamp: *The dependency pair framework: Combining techniques for automated termination proofs*. Em *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, páginas 301–331, 2004. http://dx.doi.org/10.1007/978-3-540-32275-7_21. 7

[13] Giesl, Jürgen, René Thiemann, Peter Schneider-Kamp e Stephan Falke: *Mechanizing and improving dependency pairs*. J. Autom. Reasoning, 37(3):155–203, 2006. http://dx.doi.org/10.1007/s10817-006-9057-7. 7

[14] Korp, Martin, Christian Sternagel, Harald Zankl e Aart Middeldorp: *Tyrolean Termination Tool 2*. Em Treinen, Ralf (editor): *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 de *Lecture Notes in Computer Science*, páginas 295–304. Springer, 2009, ISBN 978-3-642-02347-7. http://dx.doi.org/10.1007/978-3-642-02348-4_21. 7

[15] Krauss, Alexander: *Certified size-change termination*. Em Pfenning, Frank (editor): *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 de *Lecture Notes in Computer Science*, páginas 460–475. Springer, 2007, ISBN 978-3-540-73594-6. http://dx.doi.org/10.1007/978-3-540-73595-3_34. 7

[16] Lee, Chin Soon, Neil D. Jones e Amir M. Ben-Amram: *The size-change principle for program termination*. Em Hankin, Chris e Dave Schmidt (editores): *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, páginas 81–92. ACM, 2001, ISBN 1-58113-336-7. http://doi.acm.org/10.1145/360204.360210. 1, 6, 31

[17] Manolios, Panagiotis e Daron Vroon: *Termination analysis with calling context graphs*. Em Ball, Thomas e Robert B. Jones (editores): *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 de *Lecture Notes in Computer Science*, páginas 401–414.

Springer, 2006, ISBN 3-540-37406-X. http://dx.doi.org/10.1007/11817963_36. 2, 6, 7, 31

[18] Ramos, Thiago Mendonça Ferreira e Mauricio Ayala-Rincón: *Formalization of the Undecidability of the Halting problem for a Turing Complete Functional Language*. Em *Anais da Primeira Escola de Informàtica Teórica e Métodos Formais*, PLDI '07, páginas 165–174, Natal, Rio Grande do Norte, Brasil, 2016. UFRN. 2, 68

[19] Thiemann, René e Jürgen Giesl: *Size-change termination for term rewriting*. Em Nieuwenhuis, Robert (editor): *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 de *Lecture Notes in Computer Science*, páginas 264–278. Springer, 2003, ISBN 3-540-40254-3. http://dx.doi.org/10.1007/3-540-44881-0_19. 6

[20] Thiemann, René e Christian Sternagel: *Certification of Termination Proofs Using CeTA*. Em Berghofer, Stefan, Tobias Nipkow, Christian Urban e Makarius Wenzel (editores): *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 de *Lecture Notes in Computer Science*, páginas 452–468. Springer, 2009, ISBN 978-3-642-03358-2. http://dx.doi.org/10.1007/978-3-642-03359-9_31. 7

[21] Turing, Alan M.: *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, 2(42):230–265, 1936. http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf. 1

[22] Turing, Alan M.: *The early British computer conferences*. capítulo Checking a Large Routine, páginas 70–72. MIT Press, Cambridge, MA, USA, 1989, ISBN 0-262-23136-0. http://dl.acm.org/citation.cfm?id=94938.94952. 1, 24