UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
Curso de Sistemas de Informação
Tiago Campos Ferreira

IMPLEMENTAÇÃO DE UMA LINGUAGEM PARA VERIFICAÇÃO DE PROVAS BASEADA NA TEORIA DE TIPOS DEPENDENTES E UM PROTÓTIPO DE AUTOMATIZADOR DE PROVAS MULTI-AGENTE

Diamantina
2023

Tiago Campos Ferreira

IMPLEMENTAÇÃO DE UMA LINGUAGEM PARA VERIFICAÇÃO DE PROVAS BASEADA NA TEORIA DE TIPOS DEPENDENTES E UM PROTÓTIPO DE AUTOMATIZADOR DE PROVAS MULTI-AGENTE

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Sistemas de Informação, como parte dos requisitos exigidos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Leonardo Lana de Carvalho

Diamantina
2023

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI

FOLHA DE APROVAÇÃO

Tiago Campos Ferreira

IMPLEMENTAÇÃO DE UMA LINGUAGEM PARA VERIFICAÇÃO DE PROVAS BASEADA NA TEORIA DE TIPOS DEPENDENTES E UM PROTÓTIPO DE AUTOMATIZADOR DE PROVAS MULTI-AGENTE

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação pela Universidade Federal dos Vales do Jequitinhonha e Mucuri.

Aprovada em 06/02/2023

BANCA EXAMINADORA

Leonardo Lana de Carvalho
Faculdade de Ciências Exatas - UFVJM

João E. Kogler Jr.
Escola Politécnica - USP

gov.br Documento assinado digitalmente
JOAO EDUARDO KOGLER JUNIOR
Data: 09/02/2023 09:28:11-0300
Verifique em https://verificador.iti.br

Áthila Rocha Trindade
Faculdade de Ciências Exatas - UFVJM

sei! assinatura eletrônica

Documento assinado eletronicamente por **Leonardo Lana de Carvalho**, **Servidor (a)**, em 08/02/2023, às 10:54, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Áthila Rocha Trindade**, **Servidor (a)**, em 08/02/2023, às 11:25, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufvjm.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0977756** e o código CRC **D6A53021**.

Referência: Processo nº 23086.000729/2023-83 SEI nº 0977756

Dedico este trabalho aos meus pais e minha irmã pelo apoio incondicional.

AGRADECIMENTOS

Agradeço aos meus pais, Celso e Terezinha e a minha irmã Pâmela que me apoiaram de todas as formas possíveis e ao Arthur Maia, de modo que o apoio deles foi crucial para que este trabalho fosse concebido. Também dedico este trabalho aos meus inúmeros colegas de universidade, especialmente para a Camila e a Calliny. Ao meu orientador, Leonardo Lana de Carvalho, agradeço pela paciência e atenção nessa jornada e aos professores Áthila Rocha Trindade e João Eduardo Kögler Junior por aceitaram fazer parte desse trabalho e desempenharem um papel fundamental no desenvolvimento deste trabalho.

Finalmente, agradeço toda a comunidade da UFVJM e todos envolvidos na produção científica que indiretamente contribuíram para a produção desse trabalho.

Every propositional function $\phi(x)$ it is contended—has, in addition to its range of truth, a range of significance, i.e. a range within which x must lie if $\phi(x)$ is to be a proposition at all, whether true or false. This is the first point in the theory of types; the second point is that ranges of significance form types... (RUSSELL, 1937).

RESUMO

Técnicas formais são utilizadas para resolução de problemas, primeiro como ferramentas úteis para formalizar o pensamento humano e segundo como garantias de que nosso pensamento possa ser checado com a ajuda de uma linguagem formal. A crescente popularidade destas ferramentas tem justificativa pela necessidade de garantir a correção de provas e softwares. O conhecimento lógico-matemático serve como base para fundamentação formal. No caso que tratamos aqui, os assistentes de provas são baseados na teoria de tipos dependentes. O uso de ferramentas computacionais na concepção de assistentes de provas interativos surgiu com a necessidade de automatizar a aplicação de técnicas formais a problemas reais. Embora o uso destas ferramentas apresentem um grande avanço para a construção de processos formais orientados por computador, a sua aplicabilidade é restringida devido a limitações destas ferramentas. Uma das mais importantes limitações de assistentes de provas interativos é a incapacidade de lidar com obrigações de provas triviais, de modo que, mesmo em fatos trivialmente definidos acabam se tornando um problema de produtividade e tempo para programadores. O objetivo deste trabalho foi pesquisar soluções para este problema o que nos levou a propor o uso de técnicas de Inteligência Artificial, notadamente o uso de algoritmos multi-agente pelos quais podemos implementar processos complexos e distribuídos para a resolução de obrigações de provas em assistentes de provas baseados na teoria de tipos dependentes. Justificamos a elaboração de um novo assistente de prova baseado no cálculo $\lambda\Pi$ devido à sua simplicidade e expressividade, o que avaliamos que nos levou aos mesmos efeitos de simplicidade sobre a aplicação de algoritmos de automatização de provas na ferramenta proposta. Apresentamos assim como resultados: i) a concepção de uma ferramenta capaz de resolver provas triviais como base para o desenvolvimento futuro de um assistente de provas baseado em tipos dependentes; ii) a aplicação de um algoritmo multi-agente para a automatização de provas formais com foco em assistentes de provas baseados na teoria de tipos dependentes. Utilizando-se assim de heurísticas e de processos coletivos, concluímos que o uso de inteligência artificial em assistentes de provas é capaz de automatizar o processo de formalização, e por consequência impactar positivamente na produtividade de matemáticos e programadores. Como perspectivas futuras, deseja-se ampliar a análise de cada algoritmo e heurística sugerida através de um banco de dados de teoremas e realizar um protocolo de avaliação, a fim de confrontar os resultados para uma possível filtragem das melhores técnicas empregadas.

Palavras-chave: Teoria da Computação; Assistentes de prova; Automatização de provas; Provas Matemáticas; Tipos dependentes.

ABSTRACT

Formal techniques are used for problem-solving, first as tools useful to formalize human thought and second as guarantees that our reasoning could be checked with the help of formal language. The growing popularity of these tools is justified by the need to guarantee the correctness of software and proofs. Logical-mathematical knowledge serves as the basis for formal reasoning. In our case, the proof assistants are based on the theory of dependent types. The use of computational tools in the design of interactive proof assistants came up with the need to automate the application of formal techniques to real problems. Although the use of these tools presents a great advance for the construction of formal processes computer-oriented, their applicability is restricted due to the limitations of your tools. One of the most important limitations of interactive proof assistants is the inability to deal with trivial proof obligations. Even trivial definitions end up becoming a problem for productivity and time for programmers. The objective of this work was to search for solutions to this problem, which led us to propose the use of Artificial Intelligence techniques, notably the use of multi-agent algorithms by which we can implement distributed complex processes for the resolution of proof obligations in proof assistants based on in the theory of dependent types. We justify a new proof based on calculus λΠ due to its simplicity and expressiveness, which led to the same simplicity effects on the application of proof automation algorithms in the proposed tool. We present as results: i) the design of a tool capable of solving trivial proof as a basis for the future development of a proof assistant based on dependent types; ii) the application of a multi-agent algorithm for the automation of formal proofs with a focus on proof assistants based on the theory of dependent types. Using heuristics and collective processes, we conclude that the use of intelligence artificial proof assistants could automate the formalization process, and consequently positively impact the productivity of mathematicians and programmers. As future perspectives, it is desired to expand the analysis of each algorithm and suggested heuristic through a database of theorems to compare the results for better filtering of the best techniques applied.

Keywords: Theory of Computation, Proof assistants, Proof automation, Mathematical Proofs, Dependent types,

SUMÁRIO

1 INTRODUÇÃO 17
1.1 Objetivos 18
1.1.1 Objetivos gerais 18
1.1.2 Objetivos específicos 18

2 TRABALHOS RELACIONADOS 21

3 SISTEMAS FORMAIS 23
3.1 Modelagem formal 23
3.2 Calculo λ 24
3.2.1 Sintaxe 24
3.2.2 Estrutura do termo e substituição 24
3.3 Teoria de Tipos 27
3.3.1 Semântica 28
3.3.2 Cálculo λ Simplesmente Tipado 29
3.3.3 Lógica construtivista e Isomorfismo de Curry-Howard 29
3.4 Tipos Dependentes 30
3.5 Assistentes de provas 31

4 RECURSOS DA INTELIGÊNCIA ARTIFICIAL PARA SISTEMAS DE PROVAS 33
4.1 Algoritmo de busca 33
4.2 O problema de satisfação de restrições 34
4.3 Sistemas Multi-agente 36
4.3.1 Arquitetura de um Agente 36
4.3.1.1 Classes de agentes 37

5 IMPLEMENTAÇÃO DE UM ASSISTENTE DE PROVA 39
5.1 Métodos e Ferramentas 39
5.2 Sintaxe 41
5.3 Checador de tipos 41
5.3.1 Cálculo $\lambda\Pi$ 42
5.3.2 Símbolos Estáticos 42
5.3.3 Análise de padrões e Sistemas de Unificação 43
5.4 Implementação de um verificador de tipos 46
5.4.1 Validação de regras 48
5.4.2 Normalização 50
5.5 Resultados 51
5.5.1 Exemplos 52

6 AUTOMATIZAÇÃO 55
6.1 Síntese de programas 55
6.2 Reconstrução de provas 56
6.3 Planejadores 56
6.3.1 Automação de teorias equacionais 57
6.4 Exemplos de expansão de nós 57
6.4.1 Expansão por algoritmo evolucionário 58
6.4.2 Filtro de relevância 58
6.5 Notações 58
6.5.1 Vértices para expansão 59
6.6 MiniLambda : Um protótipo de automatizador de alta ordem . . 59
6.6.1 Representação de Termos 59
6.6.2 Representação de um problema 60
6.6.3 Arquitetura do agente 61
6.6.4 Funções de exploração 62
6.6.5 Algoritmo Multi-Agente 64
6.6.6 Resultados 66

7 CONCLUSÃO 69

REFERÊNCIAS 71

1 INTRODUÇÃO

Provas formais sempre fizeram parte do desenvolvimento da matemática, tal fato pode ser facilmente verificado observando a quantidade de teoremas construídos até a atualidade nos fundamentos da matemática vigentes. Muitos destes fundamentos estão escritos em artigos filosóficos e científicos, no conhecimento de alguns pesquisadores e por fim em computadores. Entendemos que as duas primeiras fontes de conhecimento são majoritariamente as mais ricas, no entanto, o avanço do uso de computadores para armazenar e verificar o conhecimento lógico-matemático vem chamando atenção pelas suas vantagens em relação aos métodos tradicionais (RIBEIRO; ROGGIA; VASCONCELLOS, 2021). De fato podemos observar que computadores são menos suscetíveis a erros de provas e normalmente resolvem esses problemas de forma mais rápida, isso porque as demonstrações possuem detalhes importantes, sendo que descuidos podem levar a resoluções incorretas. Ribeiro, Roggia e Vasconcellos (2021) apontam este problema ao exemplificar que muitos estudantes cometem erros na tarefa de provar a comutatividade da disjunção usando dedução natural para a lógica proposicional. Logo, a mecanização de provas pode ser uma solução para estes problemas, uma vez que garante benefícios de exatidão essenciais para lógicos, matemáticos e programadores.

O conhecimento lógico-matemático pode ser usado para a concepção e implementação de assistentes de provas. Estas ferramentas são concebidas com o objetivo de auxiliar na formalização de teorias matemáticas, sendo usados com sucesso na verificação tanto de teoremas matemáticos como de software (LEROY, 2009). Desta maneira permite-se ao usuário escrever provas formais e verifica-las através do compilador, evitando erros triviais, uma vez que o usuário pode definir propriedades lógicas e raciocinar sobre elas (GEUVERS, 2009). Com efeito, provadores apresentam algumas vantagens em relação a métodos tradicionais: primeiro porque a linguagem natural é normalmente ambígua, portanto pode ocorrer diferentes interpretações para a mesma demonstração e; segundo, pois checar uma prova pode levar apenas alguns poucos segundos, diferentemente de uma leitura extensa. Um dos pontos mais prevalentes para o uso de assistentes de provas é a possibilidade de automatizar o trabalho matemático para realizar uma demonstração formal, sendo a maioria dos trabalhos triviais, e assim normalmente ignorados por matemáticos. Logo a automatização de provas triviais possibilitaria que matemáticos focassem em problemas de maior complexidade (CZAJKA; KALISZYK, 2018). Automatizadores de provas são ferramentas para automatizar o trabalho de assistentes de provas e frenquentemente são associados com técnicas de Inteligência Artificial (IA) para a realização da busca de provas, estes automatizadores se popularizaram a partir de 1965, onde muitos métodos foram desenvolvidos graças ao uso de algoritmos baseados em inteligência artificial (GEUVERS, 2009). Dentro desse avanço podemos destacar o uso da IA como forma de

inferir termos, no caso de provas formais ou mesmo sintetizar programas. Neste trabalho iremos estudar primariamente a inferência de termos no contexto da teoria de tipos dependentes, utilizando algoritmos evolutivos como motor da nossa abordagem com inteligência artificial.

1.1 Objetivos

1.1.1 Objetivos gerais

Ao nos aprofundarmos nos tópicos de IA, o método deste trabalho foi empregar um algoritmo multi-agente para a resolução de obrigações de provas em assistentes de provas baseados na teoria de tipos dependentes.

1.1.2 Objetivos específicos

1. Analisar o estado-da-arte do uso de métodos de automatização de provas baseados em IA, notadamente em algoritmos utilizados por assistentes de provas bem conhecidos;
2. Discutir o papel da teoria de tipos dependentes aplicado na resolução de provas formais;
3. Apresentar duas implementações: a) Uma ferramenta capaz de resolver provas triviais e; b) A aplicação de um algoritmo multi-agente na automatização de provas formais.

Apresentamos assim como resultados: i) a concepção de uma ferramenta capaz de resolver provas triviais como base para o desenvolvimento futuro de um assistente de provas baseado em tipos dependentes; ii) uma aplicação de algoritmo multi-agente para automatização de provas formais com foco em assistentes de provas baseados na teoria de tipos dependentes. Utilizando-se assim de meta-heurísticas e de processos coletivos, visamos concluir que o uso de inteligência artificial em assistentes de provas é capaz de automatizar o processo de formalização, e por consequência impactar positivamente na produtividade de lógicos, matemáticos e programadores.

Primeiro, apresentaremos uma pequena revisão sobre a inteligência artificial aplicada a sistemas, primeiro formalizaremos o conceito de “problem space” e estudaremos como estes sistemas podem resolver problemas usando a inteligência artificial, sobretudo com o uso de algoritmos de busca e sistemas multi-agente para procura de soluções viáveis em problemas com grande magnitude de complexidade. Por fim, neste capítulo estudaremos o problema das restrições, pertinente para a automação de provas.

No capítulo de sistemas formais abordaremos métodos formais como forma de introduzir o leitor nas definições e notações formais que serviram de apoio para apresentar a linguagem discutida nesse trabalho. Dentro do tópico discutimos como estes sistemas podem ser modelados formalmente na área de teoria de tipos, a fim de garantir propriedades matemáticas indispensáveis para este trabalho, tais como o isomorfismo de Curry-Howard e a teoria de tipos.

No desenvolvimento desse trabalho introduziremos a implementação MPC, inicialmente abordando toda estrutura formal do desenvolvimento da linguagem. A especificação formal fica restrita primeiro pela estrutura sintática e a descrição semântica, onde descrevemos todas as regras de tipos utilizadas no núcleo do verificador de tipos da linguagem. Nossa especificação teve grande foco no problema da unificação, de modo que exploramos alguns cenários envolvidos, onde a unificação age de forma imprescindível para a verificação de provas na teoria de tipos dependentes.

Ao final desse trabalho discutiremos e formalizamos alguns conceitos-chave sobre a automatização de provas, por exemplo, os conceitos de reconstrução de provas e a expansão de nós no contexto de provas formais. Estes conceitos foram estudados a partir do ponto de vista da inteligência artificial aplicada a sistemas, complementando a discussão com o estado da arte de ferramentas de automatização de provas. Por fim, introduziremos MiniLambda, um automatizador de provas baseado em um algoritmo multi-agente. Citaremos a arquitetura em que os agentes se encontram no sistema e discutiremos o papel dos agentes no algoritmo.

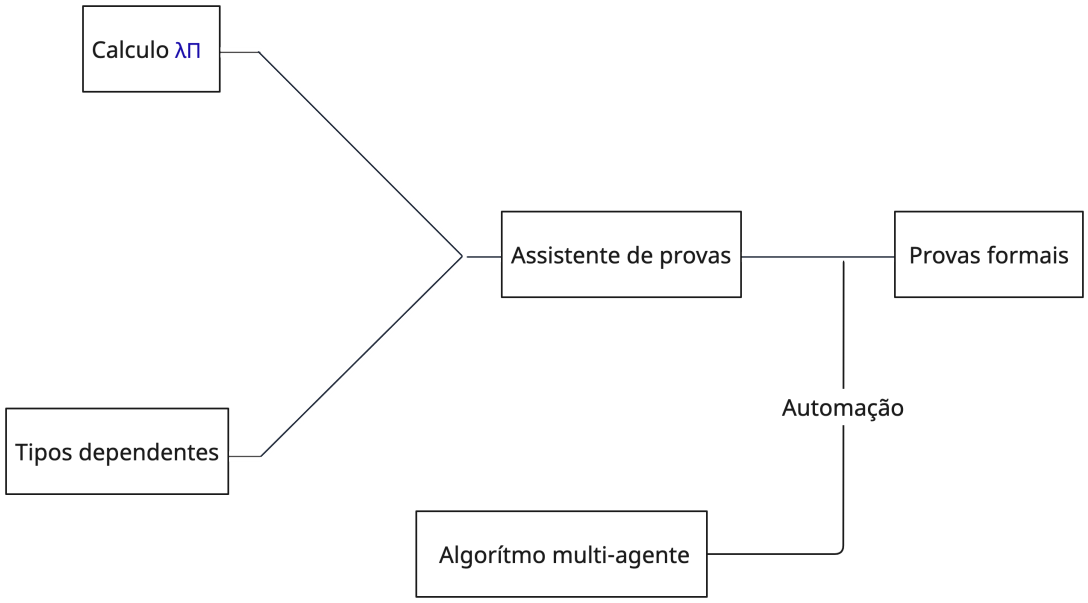

Figura 1 – Visão esquemática do procedimento da pesquisa
Fonte: Próprio Autor

Como desenvolvimento futuro sugeriremos estudar e analisar a eficácia da inteligência coletiva implementada em nosso assistente de prova para a resolução de provas triviais. Também, identificar possíveis melhorias através de testes em um banco de dados de teoremas, como realizado pelos trabalhos de Jim e Palsberg (1999) e Paulson e Blanchette (2015).

2 TRABALHOS RELACIONADOS

Existem variadas técnicas para a automação de provas em assistentes de provas na literatura científica, dentro desse grupo podemos destacar um grupo de ferramentas, os automatizadores. Na classe dos automatizadores distinguimos ainda os que utilizam métodos de aprendizagem e os que não utilizam.

A maioria das ferramentas em que há ausência de aprendizado utilizam majoritariamente de conhecimento humano para construir heurísticas e selecionar critérios relevantes para as diversas etapas de iteração, sendo consideradas ferramentas com algoritmos com poder computacional de semi-decisão para resolução de objetivos (BLANCHETTE *et al.*, 2016). Como exemplos temos: o MEPO para o assistente de prova Isabelle (MENG; PAULSON, 2009) que utiliza um filtro de seleção de teoremas baseado no compartilhamento de símbolos; o SAT solver Z3 desenvolvido pela Microsoft Research com um conjunto de teorias pré-definidas como a capacidade de verificar e analisar software incluindo softwares industriais (MOURA; BJØRNER, 2008) e a linguagem de tácticas LTAC desenvolvida para o assistente de provas Coq com o intuito de construir estratégias de provas através de definições em que o próprio programador pode prover (DELAHAYE, 2000).

Nosso trabalho vai de encontro com abordagens baseadas em meta-heurísticas e métodos estocásticos, justificamos a utilização desse tipo de abordagem, devido à ausência de trabalhos relacionados na literatura e o grande potencial destas técnicas para a automatização de provas. Yang *et al.* (2016) em seu trabalho cita a necessidade de métodos de pesquisa de provas 'out-of-box' ou seja que utilizam métodos estocásticos ou randomizados. Os autores apresentam também um método evolutivo baseado em avaliação de *fitness* para automação de provas no assistente de provas Coq. Posteriormente Nawaz *et al.* (2020) complementam dizendo que métodos tradicionais podem linearizar o processo e comprometer importantes conexões de informações. Em sequência seu trabalho descreve um algoritmo genético implementado no assistente de prova de alta ordem HOL4 com a finalidade de preservar estas conexões.

Outras ferramentas de automação de provas utilizando da abordagem de aprendizado (também conhecidas como *hammers*) capturam dados de provas anteriores com a finalidade de inferir uma possível resolução. Exemplos são o Sledgehammer (PAULSON; BLANCHETTE, 2015) do assistente de provas Isabelle e Hammer for Coq (CZAJKA; KALISZYK, 2018). Ambos automatizadores (Sledgehammer e Hammer for Coq) seguem estratégias semelhantes, primeiro os termos da linguagem são convertidos em representações de primeira ordem e depois algoritmos de *machine learning* são aplicados nesta representação, de maneira a indexar informações parciais destes teoremas e realizar a busca e a reconstrução das provas adjunto deste banco de dados.

3 SISTEMAS FORMAIS

Métodos formais são construções analíticas responsáveis por formalizar problemas e prover uma ferramenta para especificação, desenvolvimento e verificação de sistemas de informação (DéHARBE *et al.*, 2000). Tais métodos apresentam um teor de rigor capaz de ser utilizado em métodos para demonstração matemática. Tais provas formais contribuem para a confiabilidade em um projeto, propiciando também análises matemáticas apropriadas.

Uma prova matemática é a demonstração de alguma verdade reivindicada por alguma preposição. De acordo com Geuvers (2009) uma prova é absoluta no sentido de que sua validade pode ser checada por qualquer um, de modo que uma prova pode ser reduzida em partes que são testadas de maneira irrefutável. Provas formais carregam um modelo de linguagem, onde a cada passo da prova é redigida uma perspectiva semântica denominada de sistema axiomático. Portanto, uma prova pode ser reduzida à decisão do valor veritativo de uma proposição. No entanto, Zuleger (2006) defende que uma prova não é simplesmente um objetivo fixo entre um "provador" e um "verificador", mas uma interação mútua. Desse modo, para o autor, sistemas de provas se estabelecem na interação entre um possível formalizador e um analisador sem que necessariamente exista apenas uma única decisão entre certo ou incorreto.

Indubitavelmente, assistentes de provas auxiliam não apenas checando se uma prova está correta ou não. A sua grande maioria fornece comentários em relação ao progresso da demonstração. Por exemplo, no assistente de provas Coq podemos perguntar ao provador sobre a situação da prova ou procurar por algum teorema específico (PAULIN-MOHRING, 2012).

3.1 Modelagem formal

Sistemas de provas não necessariamente compartilham o mesmo entendimento em como provar uma proposição. Por exemplo, temos como contraste proposições que podem não pertencer ao conjunto de coisas prováveis em um sistema de prova, enquanto em outro sistema o conjunto mencionado é perfeitamente viável. Diferenciações de como o sistema de prova procede é comumente chamado de fundamento formal, tal conceito começou a existir a partir do ponto em que os matemáticos começaram a se preocupar com o conjunto de axiomas escolhidos para cada sistema de prova, influenciados pela abundância de questões que emergem com uma simples mudança nos axiomas (YUSHKOVSKIY; TRIPAKIS, 2018).

Sistemas formais são formados de um conjunto de axiomas A e um conjunto de regras de inferências Q, onde uma fórmula P é considerada verdadeira, caso exista uma sequência $p_1 ... p_n$, de modo que p_k seja um axioma contido em A ou uma proposição que respeite alguma regra de inferência $i_1, ..., i_n \vdash P$ em Q (YUSHKOVSKIY; TRIPAKIS, 2018).

Podemos também descrever sistemas formais definindo estados e funções de transição. Suponhamos uma dupla (D, R). D representa todas as estruturas possíveis e R é o conjunto de regras ou funções de transição de uma estrutura em D. Desta forma podemos especificar uma relação $X \rightarrow_R Y$ como uma transição de X para Y em D sobre regras de R (RICHARDSON, 2006).

3.2 Calculo λ

O que é usualmente chamado de cálculo Lambda (ou λ-calculo) é conhecido por ser uma coleção de sistemas formais baseada em notações no estilo de Alonzo Church, a partir dos trabalhos iniciados em 1930. Este sistema é basicamente uma forma de representar funções de ordem superior, onde operações ou funções podem ser combinadas para formar outras funções Hindley e Seldin (1986). A notação de Church fornece um modo simples de construção sistemática, introduzindo o símbolo λ no cálculo. Seguindo Hindley e Seldin (1986), representamos abaixo:

$$f(x,y) = x+y, \quad f^* = (\lambda x.(\lambda y.x+y)). \tag{1}$$

3.2.1 Sintaxe

Se uma expressão pertence ao cálculo lambda, podemos dizer que tal expressão é um termo λ, desta forma λx.x é um exemplo de um termo lambda que retorna o valor recebido sem fazer nenhuma modificação ao mesmo. Ainda seguindo Hindley e Seldin (1986), descrevemos a definição abaixo:

Definição Termo λ: Assume-se uma sequência infinita de expressões chamadas de variáveis, e uma sequência finita, infinita ou vazia chamada de constantes. Para os autores, um termo λ é definido indutivamente como segue abaixo.
(1) Todas as variáveis e constantes são termos λ. Também chamados *átomos*. Por exemplo, sendo x uma variável e 0 uma constante, ambos são termos λ;
(2) Uma vez que a sequência de constantes está vazia, o sistema é chamado de *puro*, caso contrário, *aplicado*. Suponha-se dois termos λ, M e N, então *(M N)* é uma aplicação;
(3) Se M é um termo λ e x é uma variável, então λx.M, é um termo λ chamado *abstração*, onde x pode ocorrer em M ou não.

3.2.2 Estrutura do termo e substituição

Nesta subseção vamos apresentar procedimentos formais para calcular com termos λ, seguindo Hindley e Seldin (1986).

Definição Variáveis atadas e livres: Uma variável x em um termo M é dita:.

- Atada, se x se ocorre do escopo de uma abstração lambda $\lambda.x$ em M;

• Livre, se x não ocorre no escopo de $\lambda.x$ em M.

Por exemplo, x ocorre livre em $(\lambda w.(\lambda z.(xz)))$, enquanto é atada em $(\lambda x.(\lambda z.(xz)))$, uma vez que aparece também no início do termo λ. (SELINGER, 2008)

De acordo com Hindley e Seldin (1986) e Selinger (2008), temos:

Definição FV(P), define o conjunto de variáveis livres em P. Por exemplo, FV((x ((λz. (λy.(my)))))) = {x, m}.

Definição Substituição: Para M, N, x, a notação $[N/x]M$ denota uma substituição. Substitui-se N por ocorrências livres de x em M. Ainda acordo com Hindley e Seldin (1986), as regras são definidas como:

$$
\begin{aligned}
&[N/x]x \equiv N;\\
&[N/x]a \equiv a,\ \forall a \not\equiv x;\\
&[N/x](PQ) \equiv ([N/x]P\ [N/x]Q);\\
&[N/x](\lambda x.P) \equiv \lambda x.P;\\
&[N/x](\lambda y.P) \equiv \lambda y.P,\ se\, x \notin FV(P) | \, x \not\equiv y;\\
&[N/x](\lambda y.P) \equiv \lambda y.[N/x]P,\ se\, x \in FV(P)\, and\, y \notin FV(P) | \, x \not\equiv y;\\
&[N/x](\lambda y.P) \equiv \lambda z.[N/z][z/y]P,\ se\, x \in FV(P)\, e\, y \in FV(P) \, | \, x \not\equiv y\, e\\
&\ z\ escolhida\, para\, ser\, a\, primeira\, varivel \notin FV(NP).
\end{aligned}
$$

A redução no cálculo λ envolve a substituição de expressões por variáveis livres. O mesmo procedimento ocorre de modo semelhante quando parâmetros em uma definição de função são passados como argumentos em uma chamada de função. Apresentamos a seguir a definição da redução β, sequindo o formalismo de Hindley e Seldin (1986).

Definição $\rightarrow_\beta$, define-se como $(\lambda x.M)N \rightarrow_\beta M[N/x]$, onde o termo $(\lambda x.M)N$ é chamado de β-redex (HINDLEY; SELDIN, 1986). Selinger (2008) cita esse processo como uma forma de conectar argumentos a funções de modo a reduzir o escopo via avaliação. É importante ressaltar que em qualquer presença de um β-redex, $\rightarrow_\beta$ reduzirá apenas o β-redex que for selecionado, de tal forma que na existência de varios β-redexes é necessario a aplicação de $\rightarrow_\beta$ para cada um sobre uma ordem de seleção a ser definido a gosto.

Definição $\rightarrow_\eta$, define-se como λx.(M x) $\rightarrow_\eta$ M (SØRENSEN; URZYCZYN, 2006), Selinger (2008) justifica a lei da η-redução com o princípio da extensionalidade,

onde duas funções que sempre retornam dois valores iguais para a mesma entrada são consideradas iguais.

Definimos a partir disso, um operador de normalização e o teorema de Church e Rosser (indicada pelo diagrama 2), é importante denotarmos a importância dessas definições, uma vez que garante propriedades importantes para a constituição desse trabalho, por exemplo, é através do processo de normalização que definimos a computabilidade dos termos utilizados neste trabalho no assistente de provas proposto, além disso, o teorema de Russer e Church, garante que a nossa ferramenta de automatização possa encontrar diversos caminhos para o mesmo objetivo.

Definição $\longrightarrow$: o operador é definido como uma *etapa*, uma etapa constitui-se de operações que devem ser aplicados a um termo λ para alcançar a sua forma básica normal.

- β-Redução : Definido com a relação $\rightarrow_\beta$
- η-Redução : Definido com a relação $\rightarrow_\eta$

Definição : Um termo está em sua forma normal, se M $\longrightarrow$ N, N = M, ou seja, não há nenhuma etapa($\longrightarrow$) restante.

Definição : Dois termos são ditos α-conversíveis ou α-equivalentes, se os dois termos são iguais em termos estruturais, λx.x é α-equivalente a λy.y, porque basta apenas renomear a variável x para y, ou y para x e obtemos o mesmo termo (SELINGER, 2008). Denotamos uma relação de igualdade entre dois termos α-conversíveis x e y como $x =_\alpha y$.

Definição : Dois termos são ditos β-conversíveis ou β-equivalentes, se dado M e N, M $\longrightarrow_n$ P e N $\longrightarrow_{n'}$ P', P é α-equivalente a P', onde n e n' são a quantidades de etapas ($\longrightarrow$) aplicadas nos termos.

Teorema (Church and Rosser) : Suponha que M e N e P são termos lambdas e M $\longrightarrow$ N e M $\longrightarrow$ P, então existe um termo Q tal que N $\longrightarrow$ Q e P $\longrightarrow$ Q (SELINGER, 2008), este teorema afirma a confluência de sistemas baseados na redução via ($\longrightarrow$). A prova pode ser encontrada nos trabalhos de Selinger (2008).

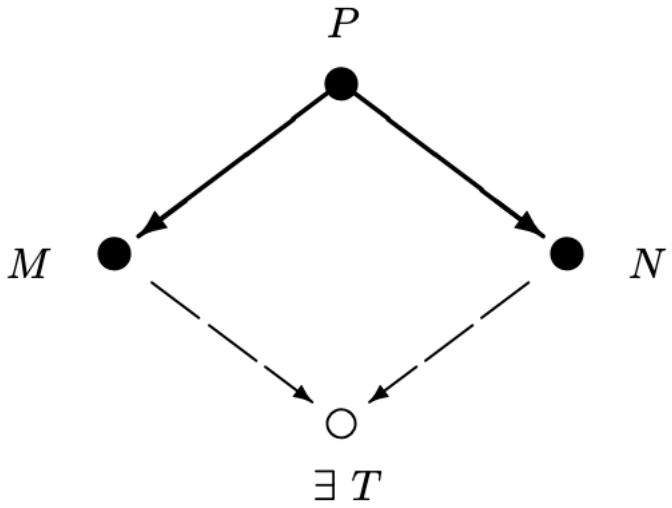

Figura 2 – Diagrama representativo do teorema de Church and Rosser.
Fonte : Hindley e Seldin (1986, p. 14)

3.3 Teoria de Tipos

A teoria de tipos foi desenvolvida como um modelo formal com capacidade de contrapor a teoria de conjuntos de Russell (TORRENS, 2019). Tipos apresentam interessantes propriedades como a sua forte relação com a lógica intuicionista e sua capacidade de integrar sistemas computacionais. Devido as suas propriedades, a teoria de tipos foi estudada e desenvolvida durante o século XX, a fim de resolver problemas relacionados à teoria da prova ou ao paradoxo de Russell (PIERCE, 2002). A partir desse ponto, a teoria de tipos se desenvolveu em diversas ramificações. Para os cientistas da computação é uma ferramenta para sistemas de tipos e para os matemáticos ela compartilha elementos entre a teoria de conjuntos e a teoria de categorias, portanto, servindo de base para diversas representações de abstrações matemáticas.

> Type theory and certain kinds of category theory are closely related. By a syntax-semantics duality one may view type theory as a formal syntactic language or calculus for category theory, and conversely one may think of category theory as providing semantics for type theory. [1]

A maioria das aplicações de teoria de tipos utilizada atualmente se concentra no uso de verificadores de tipos. O maior intuito dos verificadores é a capacidade de detectar erros de código de um programa. Em tese o programador pode confiar no seu programa a partir do ponto que o verificador de tipo comunicar que seu programa está bem tipado(PIERCE, 2002). Por exemplo, em linguagens estaticamente tipadas como Haskell, Ocaml e Clean os tipos permitem representar objetos da linguagem

[1] Fonte : https://ncatlab.org/nlab/show/relation+between+type+theory+and+category+theory

e o checador de tipo assegura que cada dado foi devidamente usado corretamente, embora comumente os fundamentos matemáticos divergem entre as linguagens.

3.3.1 Semântica

Um termo T tem um tipo t quando $T : t$ for uma preposição verdadeira. Para uma proposição ser considerada verdadeira (ou bem tipada) estabelecemos relações através de regras de inferências, de modo a confrontar tipos e termos (PIERCE, 2002).

$$\text{T-Zero}\ \frac{}{0 : Nat} \qquad\qquad \text{T-Succ}\ \frac{x : Nat,\ Succ : Nat \to Nat}{Succ(x) : Nat}$$

Regra de inferência dos números naturais .

A regra de introdução ‘T-Zero’ é um axioma que afirma que todo termo ‘0’ tem o tipo dos números naturais. ‘T-Zero’ não recebe nenhuma premissa, enquanto ‘T-Succ’ contem dois termos usados na conclusão da regra (podemos ler as notações de regras de cima para baixo).

$Succ(x)$ citado em T-Succ, é o sucessor de algum número, onde na regra ‘T-Succ’ o termo $Succ(x)$ é considerado um número natural, quanto a seu predecessor ‘x’, podemos trivialmente inferir que é também um número natural, finalmente, concluímos que $Succ$ é uma endofunção para os números naturais[2], isto é a partir, da aplicação de do termo $Succ(x)$ podemos verificar a validade da função $Succ$ e do termo aplicado x.

$$\text{T-Abs}\ \frac{\Gamma, x : A \vdash y : B}{\lambda x : A.y : A \to B}$$

Fonte: Adaptado de Pierce (2002, p.101 - 102)

Algumas vezes é conveniente representar nossas sequências de hipóteses como um ambiente Γ[3], de forma que Γ é uma função que mapeia termos para tipos $(Term \to Type)$, portanto convenientemente podemos guardar informações de termos e tipos em um contexto compartilhado entre as regras de inferências. Em ‘T-Abs’ citado nas regras de inferência dos números naturais, a vírgula adiciona a variável x ao contexto de forma que podemos diretamente inferir que o tipo de x é A (PIERCE, 2002). De maneira mais formal podemos ler nossa premissa ‘T-Abs’ como ‘Na condição de $x : A$ pertencente ao contexto Γ, logo, no contexto Γ, y tem tipo B’ (TORRENS, 2019). Por fim, nossa conclusão é que uma função lambda que recebe o termo $x : A$ e retorna um termo $y : B$ obrigatoriamente é uma função com domínio em A e contradomínio em B.

---

[2] Endofunção: função que é um endomorfismo (FRIPERTINGER; SCHöPF, 1999).

[3] Um ambiente pode ser entendido como um contexto de tipos, normalmente representado na literatura com uma letra grega maiscula

3.3.2 *Cálculo λ Simplesmente Tipado*

O cálculo simplesmente tipado é a versão do cálculo Lambda definido por Alonzo Church (1940) em sua versão com tipos simples, de tal forma que o cálculo é unicamente constituído de um conjunto de tipos atômicos (tipos primitivos) e operações que compõem tais tipos (TORRENS, 2019). Descrevemos as regras definidas para o cálculo λ simplesmente tipado como:

$$\text{T-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{T-Abs}\ \frac{\Gamma, x : A \vdash y : B}{\lambda x : A . y : A \to B} \qquad \text{T-App}\ \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash (f\,x) : B}$$

Regras do cálculo lambda simplesmente tipado.
Fonte: Adaptado de Pierce (2002, p.101 - 102)

É também comum a inclusão (no entanto, opcional) de uma regra para lidar com tipos embutidos nas regras do cálculo lambda simplesmente tipado (por exemplo, o conjunto dos naturais, booleanos, etc.).

$$\text{T-Const}\ \frac{T \text{ é um tipo} \quad x \in T}{\Gamma \vdash x : T}$$

Por exemplo, uma constante pode ser adicionada através da regra $T - Const$, onde T é o tipo da constante e x é um termo de tipo T, como a regra não apresenta nenhuma condição, é sempre possível inferir pelo contexto, que x tem o tipo T.

3.3.3 *Lógica construtivista e Isomorfismo de Curry-Howard*

Intuitivamente é possível identificar que ao construir programas fazemos um paralelo com algum tipo de lógica, e de fato ao trabalhar com um paradigma de programação específico estamos representando um sistema formal distintivo, por exemplo, um programador que utiliza do paradigma de programação imperativa usa como base a máquina de Turing e a máquina de registradores. Peter Landin observou essas relações ao perceber que computações complexas de linguagens podem ser formalizadas através de um conjunto de regras reduzido, o que pode ser entendido como o núcleo formal da linguagem, como exemplificado acima (PIERCE, 2002).

Da mesma forma, intuitivamente podemos perceber que ao descrever programas a partir de tipos podemos representar algum tipo de construção lógica, por exemplo, suponha-se esse programa escrito em *Python*:

```
1 def modus_ponens(f : A -> A, a : A) -> A :
2     return f(a)
```

Mais precisamente esta construção lógica é fomentada pela computabilidade dos termos. Sørensen e Urzyczyn (2006) complementa que estes λ-termos tipados correspondem a uma proposição da lógica intuicionista. Geuvers (2009) formaliza essa correspondência, conhecida como isomorfismo de Curry-Howard, onde T é uma relação de isomorfismo entre fórmulas e tipos e M é um λ-termo do mapeamento de ϕ para um tipo qualquer $T(\phi)$.

$$\Gamma \vdash_{logica\,intuicionista} \phi \; se\,e\,somente\,se\, \Gamma' \vdash_{teoria\,de\,tipos} M : T(\phi)$$

3.4 Tipos Dependentes

Teorias baseadas em tipos dependentes surgiram em torno dos anos 70, extremamente influenciadas pelos trabalhos de Martin-Lof (NORELL, 2007). Estas teorias criam conexões com a lógica intuicionista e o isomorfismo de Curry-Howard, tanto que atualmente serve como base para uma abundância de assistentes de provas, como o descrito neste trabalho.

De forma simples, tipos dependentes significam que tipos podem se referir a outros tipos, e podem ser tratados como valores comuns, isto é são tipos de primeira classe (ABEL, 2009), não obstante, eles generalizam tipos de forma que podemos expandir para a noção de família de tipos (TORRENS, 2019), aonde a partir do valor de um tipo dependente pode-se criar uma infinidade de comportamentos para a mesma abstração de tipo.

O exemplo mais frequente é o tipo Π conhecido como projeção ou tipo de sigma (TORRENS, 2019).

$$\Pi - \frac{\Gamma \vdash A : s \qquad \Gamma, (x : A) \vdash B : s}{\Gamma \vdash \Pi x : A.B : Type}$$

Regras de inferência do tipo Π Fonte: Adaptado de Saillard (2015, p. 32)

No caso da regra acima representado na regra acima, Π é uma construção onde x pode depender de B ou não, podemos ler como para todo x, em que B(x) é verdade. Um exemplo frequente são vetores, embora sejam semelhantes a listas, eles são frequentemente indexados pelo seu tamanho, por exemplo, considere este código abaixo em Python:

```
1  def imprima() :
2      vec = [1]
3      print(vec[0])
4      vec = []
5      print(vec[0])
```

No código é requerido que imprima o valor na posição 0, no entanto, para esta posição no segundo vetor não existe valor, acarretando um erro. Com tipos depen-

dentes podemos abstrair este problema com um vetor em que seu tamanho esteja anotado em seu tipo. Considere $f : \Pi n : \mathbb{N}.Vec\,n$, então $f(10)$ irá sempre produzir um vetor com 10 elementos (TORRENS, 2019). Da mesma forma podemos produzir uma função que extraia um valor apenas de vetores que não são vazios $f : \Pi n : \mathbb{N} \rightarrow Vec(n+1) \rightarrow T$, onde T é o tipo do elemento contido no vetor.

3.5 Assistentes de provas

Assistentes de provas são a última peça de sistemas de provas, uma vez que eles mecanizam elementos formais para a checagem em computador. Dixon e Fleuriot (2006) cita os casos de uso de assistentes de provas para construção de softwares verificados, uma vez que, contrariamente a teoremas matemáticos, softwares são inviáveis de serem verificados por métodos tradicionais, devido à grande complexidade e tamanho da sua formalização.

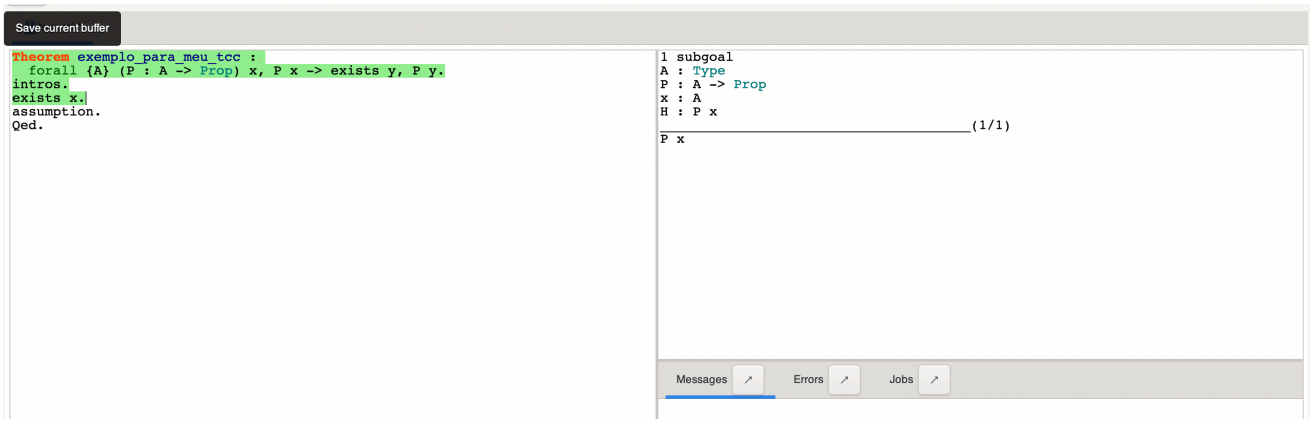

Figura 3 – Screenshot do assistente de prova Coq em modo de formalização.
Fonte: Próprio Autor

A prova definida acima na figura 3 é um exemplo de um famoso assistente de prova que pode ser utilizado para raciocinar formalmente. Cada etapa em linha verde significa o progresso da prova e o quadrado a direita significa o contexto da prova. No exemplo, Geuvers (2009) caracteriza o estilo da prova como procedural, isto significa que embora o leitor consiga identificar pontos-chave para entender como foi provado o teorema, a construção da prova em si é feita com o único objetivo do computador realizar a prova. Portanto, não é o objetivo agradar o leitor com legibilidade. Caso este seja o objetivo, Geuvers (2009) nomeia o estilo de prova como declarativo, neste caso provas são de mais fácil leitura como no caso de artigos matemáticos.

4 RECURSOS DA INTELIGÊNCIA ARTIFICIAL PARA SISTEMAS DE PROVAS

O uso de IA para sistemas se tornou popular para diversos propósitos, em destaque para a resolução de problemas difíceis, otimização ou a interação entre humano e computador. Langley e Laird (2006) menciona este fenômeno ao destacar a transição entre 1970s e 1980s sobre o rumo em que a inteligência artificial tomaria, onde a visão original de uma inteligência artificial com propriedades humanas foi modelada para diversas outras áreas de pesquisas como estas citadas acima. Os problemas de busca são frequentemente associados a grande complexidade e uma vasta possibilidade de "caminhos". Korf (1996) descreve mais formalmente problemas de busca através de um modelo chamado de "problem space", onde o ambiente de um problema de busca pode ser descrito como uma tupla $(S_0...S_n,\ f : S_n \longrightarrow S_{n+1})$, onde S é o conjunto de estados do problema e f é o conjunto de operações que podem modificar o estado S_n, no qual o índice n representa o estado atual.

4.1 Algoritmo de busca

Problemas de alta complexidade frequentemente requerem um algoritmo capaz de escolher uma solução adequada entre um conjunto de soluções. Não obstante, algumas destas soluções podem ser dificilmente alcançadas graças à limitação computacional. De modo a resolver estes problemas computacionais, o uso de inteligência artificial adjunto com algoritmos de busca são aplicados para a resolução destes problemas.

Quando empregamos sistemas de buscas para a resolução de um problema, raramente conhecemos o custo/quantidade de caminhos que o algoritmo irá requisitar para identificar uma solução ideal. Desta forma estes algoritmos normalmente buscam através de tentativas e falhas, realizando um procedimento de exploração (KORF, 1996). Russell e Norvig (2010) e Alves (2014) descreve este problema classificando o custo por 4 propriedades, a sua completude, isto é, se o método consegue encontrar uma solução, complexidade de tempo de execução do programa, complexidade de espaço na memória, e por fim sua capacidade de encontrar soluções com alto nível de qualidade em cenários de diferentes tipos de soluções.

Durante o processo de busca várias soluções são consideradas, de modo que o conjunto de soluções podem mudar conforme a evolução do algoritmo, uma vez que cada algoritmo se utiliza de uma estratégia diferente para a busca de um novo estado, Russell e Norvig (2010) descreve dois tipos de estratégias para a busca. Por exemplo, algoritmos baseados em busca não-informada são incapazes de escolher um novo estado baseado em algum tipo de conhecimento, por outro lado, um algoritmo de busca-informada realiza suas escolhas baseado em algum tipo de informação. Uma heurística então se define por uma decisão baseada em um parâmetro conhecido. Bandaru e Deb (2016) define heurística como uma forma de resolver um problema em que o resultado pode ser uma aproximação, e não há necessidade da prova da convergên-

cia matemática, por fim não obrigatoriamente o algoritmo precisa procurar em cima de todas as soluções possíveis para encontrar uma resposta, portanto o algoritmo garante eficiência computacional. Por consequência, buscas informadas conseguem gerar resultados mais satisfatórios do que buscas não-informadas (RUSSELL; NORVIG, 2010).

As classes de problemas de busca mais conhecidas são a busca de caminho mínimo, jogo de disputa, e por fim o problema de resolução de satisfação de restrições (KORF, 1996). Alguns exemplos são o clássico problema do caixeiro viajante(busca de caminho mínimo), jogo de xadrez (jogo de disputa) e finalmente o problema da satisfatibilidade booleana (satisfação de restrições).

É notável a presença de trabalhos relacionados a resolução de satisfação de restrições em áreas da teoria de tipos, um exemplo é o uso de algoritmos de resolução de inferência de tipos como nos trabalhos apresentados por (CHANDRA *et al.*, 2016) e (JIM; PALSBERG, 1999). É fundamental o uso de IA para inferência de tipos, o que é justificado pela natureza não-decidível de alguns sistemas de tipos. Embora nem todo tipo de inferência de tipos seja indecidível, como no caso de sistemas baseados em Hindley Milner (SULZMANN, 2001), para a maioria é impossível de se construir um programa capaz de inferir qualquer tipo de termo (DUNFIELD; KRISHNASWAMI, 2020).

Os trabalhos de Nawaz *et al.* (2020) e Yang *et al.* (2016) são um exemplo do uso de heurísticas e eventos estocásticos para resolução de provas em assistentes de provas de alta ordem. Isto é, recorrer a eventos indeterminativos tais como aleatoriedade para procurar soluções em assistentes de provas que suportam quantificadores universais ($\forall$) e existenciais ($\exists$). Embora o uso de eventos estocásticos para automatização de provas seja algo pouco explorado (NAWAZ *et al.*, 2020), Yang *et al.* (2016) argumenta que o uso de algoritmos baseados em eventos estocásticos é promissor e com grande potencial de influência na automatização de provas.

4.2 O problema de satisfação de restrições

Um problema de satisfação de restrições (CSPs da sigla em inglês) é um problema definido por um conjunto de variáveis $x_1 ... x_n$ e um conjunto de restrições $C_1 ... C_n$, onde cada variável x_i é definida por uma restrição em C_i em que restringe o valor de um conjunto de variáveis em x (RUSSELL; NORVIG, 2010). Uma solução consistente é um conjunto de valores que relaciona cada variável x para um valor que respeita todas as restrições em C.

Exemplos de problemas de satisfação de restrições são recorrentes em problemas de criptografia. Brailsford, Potts e Smith (1999) exemplificam o problema de cifra de substituição. Um alfabeto é formado de um léxico e cada letra tem um significado semântico que pode ser descoberto através de um conjunto de restrições. Não obstante, problemas de satisfação de restrições são comumente vistos em outras áreas

da computação, inclusive para resolução de problemas de inferência de tipos. Nos trabalhos de Jones (2019), ele apresenta como o problema de satisfação de restrições é utilizado na linguagem Haskell para inferir tipos de programas.

Por exemplo, suponha-se um programa :

```
1  sum x y = x + y
```

O programa acima demonstra o problema de descobrir o tipo das variáveis x e y e da função sum, Jones (2019) descreve este tipo de problema como um problema de CSP. Primeiro os tipos das variáveis são generalizados, x é α e y é Σ, após isso as restrições são construídas a partir do programa, $\alpha \sim$ Number, e no que lhe concerne $\Sigma \sim$ Number, e por fim se aplica um solucionador de restrições.

Outros exemplos são linguagens de programação baseadas em restrições, como no caso da linguagem de programação Oz. Suponha-se, que seu objetivo seja encontrar o valor de uma equação $SEND + MORE = MONEY$, onde cada letra representa um dígito e com a restrição que S e M sejam diferentes de 0.

```
1  proc {Money Root}
2     S E N D M O R Y
3  in
4     Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)        % 1
5     Root ::: 0#9                                       % 2
6     {FD.distinct Root}                                 % 3
7     S \=: 0                                            % 4
8     M \=: 0
9                  1000*S + 100*E + 10*N + D             % 5
10    +            1000*M + 100*O + 10*R + E
11    =: 10000*M + 1000*O + 100*N + 10*E + Y
12    {FD.distribute ff Root}
13 end
```

Fonte :
<http://mozart2.org/mozart-v1/doc-1.4.0/fdt/node15.html#section.problem.money>

Em programação lógica de restrições cada componente básico do programa se comporta como uma restrição. Assim sendo, o fluxo do programa se constrói por um conjunto de restrições chamado de regras (JAFFAR; LASSEZ, 1987). No contexto da linguagem de programação OZ, emprega-se um resolvedor de restrição conhecido como Mozart. Este verifica cada restrição e a associa a um procedimento conhecido como *propagator*. Por fim estes *propagators* irão reduzir o problema ao conjunto de possíveis variáveis da solução, a fim de achar a solução do problema ou refiná-la para outro *propagator* (SABOGAL *et al.*, 2013).

4.3 Sistemas Multi-agente

Sistemas Multi-agente (SMA ou, *MAS* em inglês) é uma importante área de sistemas inteligentes. *MAS* provém uma forma de organização em que múltiplas interações através de agentes autônomos são compostas em um espaço computacional, de forma onde a colaboração satisfaça algum objetivo (WOOLDRIDGE, 2009).

É importante destacar que é a definição de sistemas multi-agente pode ser conflituosa com a noção de inteligência artificial distribuída. Este ramo da IA lida com a resolução de problemas dividindo as tarefas por sistemas cooperativos de solucionadores, onde cada solucionador compartilha recursos computacionais ou um conhecimento parcial da sua solução (SARMA, 2011), enquanto sistemas coletivos de inteligência lidam com a noção de grupos de individuais agindo coletivamente para um objetivo comum (MALONE; BERNSTEIN, 2015), por exemplo, no caso da inteligência coletiva baseada em *swarm*, o papel de cada unidade tem significado na auto-organização, onde cada partícula do sistema lida de forma auto-organizada para resolver um certo problema (DEEPA; SENTHILKUMAR, 2016).

Sistemas multi-agente são comumente usados para resolução de problemas de otimização, de decisão, modelagem e simulação (SGHIR, 2016). Isto se dá pelo poder de paralelização e colaboração que ocorre em sistemas constituídos por agentes. Durfee e Rosenschein (1995) argumentam que o princípio da benevolência dos agentes alimenta a contribuição para que o objetivo seja alcançado. Uma vez que todos os agentes desejam o mesmo objetivo, as tarefas podem ser delegadas, aumentando consideravelmente as chances de sucesso. Estes papéis de benevolência podem ser exemplificados com um conjunto de regras bem estabelecidas. Ferber, Gutknecht e Michel (2008) citam este fenômeno ao descrever o conceito de *roles*(papéis). Um papel é um conjunto de restrições (obrigações, requisitos e habilidades) que um agente precisa obedecer. Além disso, papéis também propagam qual o conjunto de padrões de interações entre os agentes o sistema irá estabelecer.

4.3.1 Arquitetura de um Agente

De acordo com Russell e Norvig (2010), um agente é uma unidade que age de alguma forma, onde esta ação deve ser orientada por um conjunto de sensores em um ambiente. Além disso, esta ação deve ser autônoma, de forma que este agente possa ter a habilidade de coabitar em um ambiente, perceber este ambiente, se adaptar, de modo a perseguir seus objetivos, finalmente, o agente deve se orientar a sua ação de maneira autônoma, isto é deve se basear em seu conhecimento construído a partir do aprendizado parcial, ou mesmo incorreto que lhe foi dado inicialmente (RUSSELL; NORVIG, 2010).

Wooldridge (2009) apresentam uma formalização de um agente simples. Primeiro suponha:

$$\text{Ac} = \{\alpha, ...\}$$

$$E = \{e_0, ..., e_u\}$$

Sendo Ac como uma lista finita de ações de um agente, e E um conjunto de estado, defini-se *run*, como uma lista interligada de ações e estado denotada por $(r : e_0, ...)$, logo:

- R é a combinação de todas as sequências entre Ac e E.
- R_{Ac} é subconjunto de R que termina com uma ação.
- R_E é subconjunto de R que termina com um ambiente E.

Um efeito de uma ação de um agente em um ambiente é então formalizado como uma função transformadora de estado $R_{ac} \rightarrow P(E)$, ou seja, esta função mapeia um conjunto R que termina em uma ação para um conjunto de novos estados do ambiente. É importante ressaltar que P é uma função não-determinística.

Um ambiente é representado por uma tripla (E, e_0, Γ), onde E é um conjunto de ambientes, e_0 é um estado inicial em E, e Γ é uma função transformadora de estado. Posteriormente podemos formalizar um agente como uma função $Ag : R_E \rightarrow Ac$, portanto sendo uma função com a responsabilidade de escolher uma nova ação a partir de um histórico de ações e ambientes transformados por estas ações.

$$R(Ag, Env) = (e_0, a_0, e_1, a_0, ...)$$

Finalmente Wooldridge (2009) formaliza a concepção de um agente inserido em um ambiente como $R(Ag, Env)$ sendo uma sequência de ações e estados, tais que:

- e_0 é o estado inicial;
- $e_0 = Ag(e_0)$;
- para todo u, $a_u = Ag(a_0, e_u, ...a_u)$;
- para todo u, $e_u \in \Gamma((e_0, a_0, e_u))$;

4.3.1.1 Classes de agentes

Weiss (2000) argumenta a necessidade de uma descrição para os diferentes tipos de agentes, como na função Ag formalizada na arquitetura do agente, precisamos especificar que tipo de ação será realizada a partir de um agente. Weiss (2000) cita estas classificações como:

- Agente lógico, cuja decisão é baseada em um aparato lógico-dedutivo.
- Agente reativo, cuja decisão é diretamente acionada por um mapeamento de uma situação para uma ação.
- Agente *Belief–Desire–Intention* ou agente BDI, cuja decisão é baseada em uma estrutura representando crenças, desejos e intenções.
- Agente baseado em arquitetura de camadas. Nesta arquitetura as decisões são tomadas por várias camadas de *software*, onde cada uma destas camadas realiza uma forma de processamento em diferentes níveis.

Agentes não necessariamente tem o método de iteração bem definido, por exemplo, a função Ag pode afetar indiretamente o ambiente e da mesma maneira prover um canal de comunicação entre agentes. (WEYNS *et al.*, 2004) argumenta que este modelo é caracterizado por pouco acoplamento, de forma que os agentes não precisam se conhecer explicitamente, existir no mesmo espaço, ou sequer precisam co-existir durante o mesmo período. Durante a comunicação em sistemas distribuídos e dinâmicos, agentes conseguem realizar comunicações robustas usando apenas abstrações. Este comportamento é extremamente relevante para este trabalho, uma vez que utilizaremos fundamentos baseados na comunicação indireta através de agentes lógicos distribuídos. Não obstante, nossa arquitetura multi-agente provem um conjunto de ações baseado no cálculo λ, onde cada agente pode depender de outra entidade, no entanto, preservando a habilidade de agir paralelamente.

5 IMPLEMENTAÇÃO DE UM ASSISTENTE DE PROVA

Neste capítulo apresentaremos inicialmente as escolhas para a especificação do projeto para o nosso assistente de prova. Seguidamente apresentaremos detalhes de implementação adjunto da explicação formal dos componentes do sistema de tipo, por fim estudaremos casos de usos de programas e provas que podem ser codificados em nossa linguagem. O assistente de provas foi implementando na linguagem de programação Haskell, com algumas poucas dependências, distribuído através do gerenciador de pacote *cabal*. Suas principais dependências são o *parsec* como forma de *parsing* monádico e a biblioteca contêineres para representação de busca de forma otimizada.

Neste trabalho chamaremos de MPC (*Minimal Proof Checker*) nosso analisador de provas a ser desenvolvido e estudado nos próximos capítulos. A fim de especificar algum conceito do MPC podemos nos referir a ele tanto como uma linguagem de programação quanto como assistente de prova [1]. O MPC pode ser definido como uma linguagem de programação funcional com suporte a tipos dependentes, sistema de unificação e suporte para automação através de busca informada. Sua principal motivação é providenciar uma ferramenta de desenvolvimento de especificações formais de programas com suporte a automatização de provas, mantendo simplicidade e minimalidade na sua implementação.

5.1 Métodos e Ferramentas

O sistema deve ser capaz de receber um *input* (entrada), nesse caso um arquivo de texto MPC. Esse arquivo irá passar por diversas etapas, dessa forma, caso o *input* recebido seja valido, o interpretador irá mandar uma mensagem certificando a validade do programa, ou alternativamente, o programa irá parar sua execução caso tenha detectado um erro durante o processo.

Existem diferentes tipos de erros que podem parar a execução do programa. Podemos definir cada processo pelo fluxograma abaixo:

[1] Como explicado em capítulos anteriores, ambos conceitos (linguagem de programação e assistente de provas) podem ser equivalentes

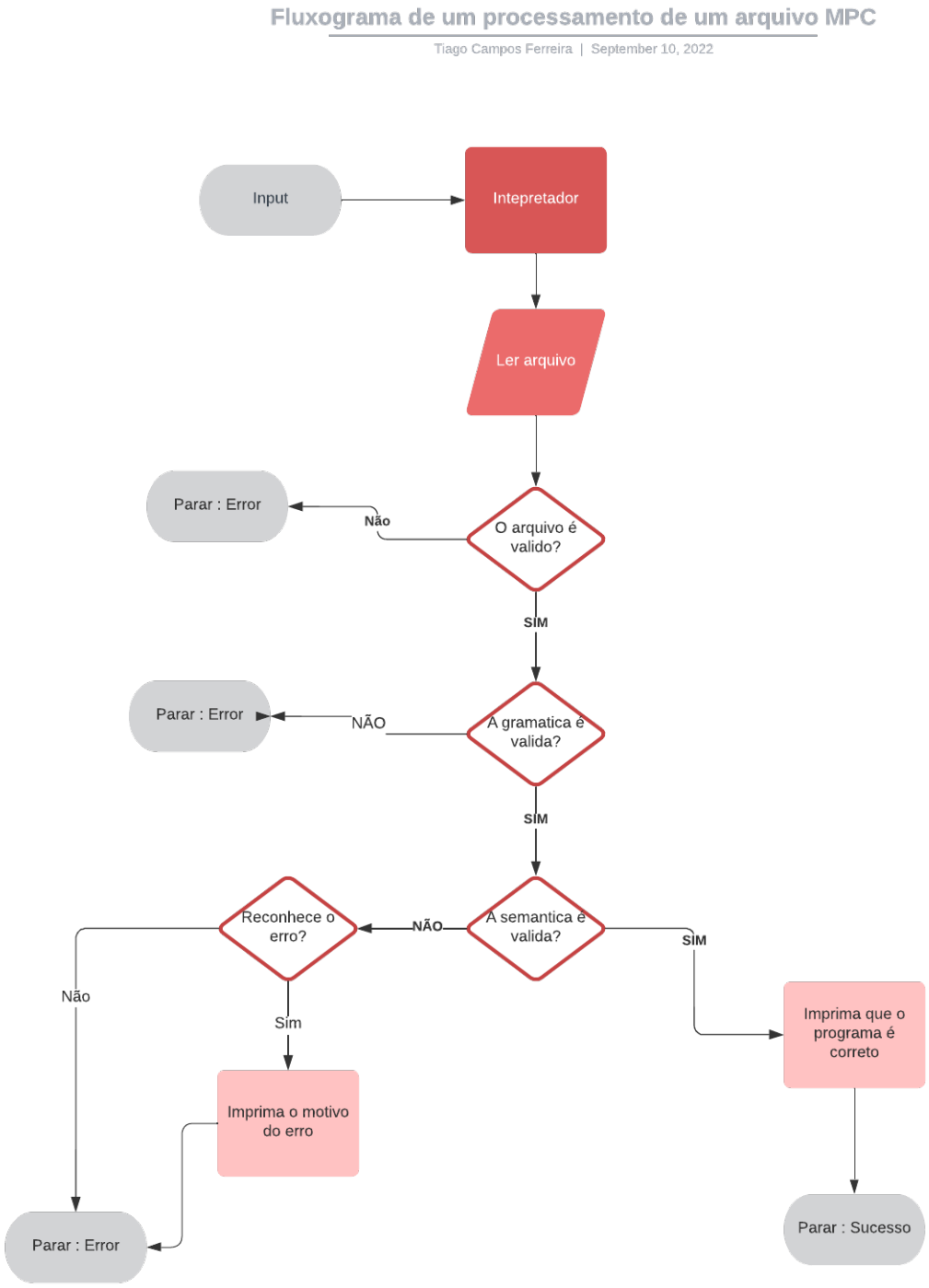

Figura 4 – Fluxograma de um processamento para o interpretador MPC
Fonte : Próprio Autor

5.2 Sintaxe

Descrevemos a sintaxe do MPC abaixo, um termo m é constituído de :

$$m ::= x \text{[Variável]}$$
$$(f\,x) \text{[Aplicação]}$$
$$(x : A) \sim> B \, ou \, A \sim> B \text{[Tipo]}$$
$$|\,x...x' :: A => b \text{[Lambda]}$$
$$\{m :: |C...C'\} \text{[Construtor Opcional]}$$
$$x\,of\,T\,[\,|P => B \,...\, |P' => B'] \text{[Analisador de padrões]}$$
$$def\,n = b; b' \text{ se n aparece em b' então n <- b, [Def]}$$

Nestes termos, $x..x'$ é uma lista de variáveis contendo pelo menos uma variável. $|P => B \,...\, |P' => B']$ e $C...C'$ é uma lista de padrões que pode ser vazia. Além disso, (f x y) significa o mesmo que ((f x) y).

Por fim temos a sintaxe de definições globais e símbolos estáticos.

$$\text{Definição } D\,m ::= Static\,S : m \text{[Simbolo Estático]}$$
$$D\,m \text{[Definição global]}$$

Sendo m um termo e D um nome que contém qualquer carácter exceto ':', '(', ')', '.', '|', ' ', '>', '', '', '=', '[', ']', ';'

```
Static or : ~ * ~> ~ * ~> *.
Static or_right : (x : *) (y : *) ~ x ~> (or x y).
Static or_left : (x : *) (y : *) ~ y ~> (or x y).
Or
  |A B :: ~ * ~> ~ * ~> * => {(or A B) :: |or_right | or_left}.
```

O exemplo acima é uma demonstração de sintaxe valida em MPC.

5.3 Checador de tipos

Nosso verificador de tipos é baseado na teoria de tipos dependentes. Em resumo, os mesmos tipos de linguagens de programação comuns, como C++, podem depender de outro valor. Por exemplo, um tipo de um vetor V pode depender de um número natural N que é seu tamanho, construindo uma projeção de tipo $\Pi\, len : Nat \rightarrow V(len)$, de tal modo que um vetor [] tem o tipo $V(0)$ e um vetor não-vazio $[v_1, ..., v_n]$ tem o tipo de $V(n)$. Em nossa sintaxe qualquer projeção corresponde a $(x : A) \sim> f(x)$, sendo $f : A \sim> Type$ ou $f : A \sim> Set$, além disso, a sintaxe $\sim A \sim> B$ corresponde a uma projeção que não dependente do seu argumento $\Pi\, x : A \rightarrow B$, isto é qualquer função comum.

Primeiro apresentaremos a nossa fundamentação teórica, isto inclui o cálculo de $\lambda\Pi$, nosso sistema de símbolos e uma descrição do algoritmo de unificação de termos. Por fim, apresentaremos uma pequena introdução da implementação do núcleo do assistente de provas MPC.

5.3.1 Cálculo $\lambda\Pi$

O cálculo $\lambda\Pi$ é um sistema formal dependentemente tipado que permite fazer provas simples de lógica de primeira ordem (COUSINEAU; DOWEK, 2007). Em nosso trabalho utilizamos uma versão reduzida do $\lambda\Pi$, o que convenientemente nos garantiu um núcleo reduzido devido à simplicidade formal do cálculo. Descrevemos o cálculo $\lambda\Pi$ estendido abaixo:

$$\text{Universo}\ \frac{}{\Gamma \vdash Type : Kind} \qquad \text{Conjunto*}\ \frac{}{\Gamma \vdash Set : Type}$$

$$\text{Variavél}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\text{Aplicação}\ \frac{\Gamma \vdash f : \Pi y : A.B \qquad \Gamma \vdash x : A}{\Gamma \vdash (f\,x) : B[y/x]}$$

$$\lambda\text{-Abstração}\ \frac{\Gamma, (x : A) \vdash t : B \qquad \Gamma \vdash \Pi x : A.B : s \qquad s \in \{Type, Set\}}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}$$

$$\Pi\text{-Produto}\ \frac{\Gamma \vdash A : s \qquad \Gamma, (x : A) \vdash B : s \qquad s \in \{Type, Set\}}{\Gamma \vdash \Pi x : A.B : Type}$$

Conjunto de regras básicas para o núcleo de MPC
Fonte: Adaptado de Saillard (2015, p. 32)

5.3.2 Símbolos Estáticos

Símbolos estáticos representam a forma de codificação de dados. Neste sentido temos o construtor opcional e o construtor sub-tipagem. Descrevemos as regras de sub-tipagem e símbolos como:

$$\text{Construtor Opcional}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash c \in \theta, c : \sigma \to s \qquad s \in \{Type, Set\}}{\Gamma \vdash \{A :: \theta\}}$$

$$\text{Construtor Sub-tipagem}\ \frac{\forall c \in \theta, c \in \theta'}{\Gamma \vdash \{A :: \theta\} <: \{A :: \theta'\}}$$

Regras do sistema de símbolos e sub-tipagem

Um telescópio θ representa uma sequência de termos e tipos, onde $\theta = (x : A) .... (x' : A')$ pode ser uma lista vazia (NORELL, 2007). Aqui estendemos a definição para aplicações e λ-abstrações, portanto quando referimos $f\theta : X$ temos $f : (x : A) .... (x' : A') \to X$. Cada tipo de construtor em θ é comparado pela nomeabilidade do termo, isto é, pelo nome do termo. Considere então n sendo o tamanho de θ onde este pode ser um número natural qualquer, e i, de tamanho $0 <= i <= n$. θ_i representa então a i-ésima instância da sequência de θ, por exemplo, sendo $\theta = (A : T)(B' : T')$ e n = 2, θ_1 é $(A : T)$ e θ_2 é $(B' : T')$. Da mesma forma $typeof\ \theta_i$ indica a i-ésima instância, no entanto, apenas como a instância do tipo, por exemplo, $typeof\ \theta_1$ é A, enquanto $typeof\ \theta_1$ é B.

As regras de sub-tipagem para cada lambda termo foram omitidas, por exemplo $\{A :: \theta\} <: \{A :: \theta'\}$, então $\lambda x.x : \Pi\{A :: \theta\} \to C <: \lambda y.y : \Pi\{A :: \theta'\} \to C'$ é derivável.

5.3.3 Análise de padrões e Sistemas de Unificação

A análise de padrões é um atributo de linguagens de programação para realizar a análise de caso, isto é, eliminar valores e construir novos valores a partir disso. Uma definição de análise de padrões corresponde a um conjunto de cláusulas que algum tipo de dados precisa corresponder (COCKX, 2017).

```
1  ALGORÍTMO
2  PROCEDIMENTO eliminar_natural(n : Natural)
3    se número == 0 então
4      imprima "Número igual a 0"
5    senão
6      imprima "Número maior que 0"
7  FIM PROCEDIMENTO
8  FIM ALGORÍTIMO
```

No caso dos números naturais temos dois casos especiais, o primeiro quando o valor é 0 e o segundo quando o valor é sucessor de algum outro número natural. Da mesma forma, a análise de casos mais a recursão é correspondente à indução matemática.

```
1 prova_x+0=0
2   | x :: (x : Nat) ~> (Eq Nat x (x + 0)) => [x of (Eq Nat x (x + 0))
3     |0 => (refl nat 0)
4     |(+1 x') => (cong nat x' (+ x' 0) nat +1 (prova_x+0=0' x'))
5    ].
```

O código acima escrito em MPC faz o uso dos princípios de análise de casos e recursão para provar que algo somado a 0 sempre será ele mesmo. Primeiro analisamos o caso quando x é igual a 0, o que nos dá 0 = 0. Trivialmente chamamos reflexividade. No segundo caso, onde x é maior que 0, temos o valor (x' + 1) = x e a

hipótese de indução x' = x' + 0, por fim basta incrementar nos dois lados da equação por congruência para obter x ' 1 = x' + 1 + 0, logo temos x = x + 0. Seque abaixo regras de análise de padrão do MPC.

$$\text{Universos}\ \frac{}{s \in \{Set, Type\}}$$

$$\text{Cláusulas}\ \frac{\Gamma \vdash v : \{A :: \theta\} \qquad \Gamma \vdash \forall\,((c\,\sigma), r) \in P, (c\,\sigma) : A}{\Gamma \vdash MATCH(v, R, P) : R}$$

$$\text{Padrão}\ \frac{\Gamma, v : \{A :: \theta\}, \forall c : \sigma \to s \in \theta, \forall\,(x : X) \in \sigma, x : X \vdash [(c\,\sigma)/v]) : R}{AGAINST(v, r[(c\,\sigma)/v]) \in MATCH(v, U, R, P)}$$

Conjunto de regras básicas para a análise de padrões no MPC

$MATCH$ é uma quádrupla formada pelo valor a ser analisado, seu tipo. O novo tipo da expressão a ser analisada, uma lista de cláusulas construída por pares $((c\theta), r)$, onde c é um símbolo estático chamado de predicado. Por exemplo, $MATCH(x, Nat, (x = x + 0), [(0, (refl\ nat\ 0)), ((+1\,x), (cong\ \theta))])$ corresponde à análise de padrão do exemplo acima.

$AGAINST$ é um par de cada cláusula $MATCH$, onde $r[(c\,\theta)/v]$ corresponde ao termo r, onde toda instância de $(c\,\theta)$ é substituída por v.

No exemplo acima é notável que mesmo após analisarmos cada caso e obter a hipótese de indução ainda requisitamos a igualdade $(x' + 1) = x$. Sem essa informação ficaríamos parados na prova, decidir quando dois tipos podem ser iguais é um importante papel para sistema de provas (TORRENS, 2019), como o MPC. Cockx (2017) complementa que este processo de igualdade entre cláusulas é chamado de unificação, onde o lado à direita da equação necessita ser substituído pelo lado direito da equação, de modo a se tornar igual por definição. A partir dos trabalhos de Cockx (2017) e Norell (2007), propomos um sistema de unificação para o MPC, representado abaixo:

$$\text{Contexto Vazio } \frac{isEmpty(\Delta)}{\Delta\text{Válido}}$$

$$\text{Introdução de substituição } \frac{\Gamma \vdash U : s \qquad \Gamma \vdash U' : s \qquad \Gamma, a : U' \vdash b : U \qquad s \in \{Type, Set\}}{\{a \mapsto_U b\} \in \Delta}$$

$$\text{Anti-Reflexividade } \frac{a \neq b}{\{a \mapsto_U b\} \in \Delta} \qquad \text{Anti-Simetria } \frac{\{b \mapsto_U a\} \notin \Delta}{\{a \mapsto_U b\} \in \Delta}$$

$$\text{Unificação Simples } \frac{\{v \mapsto_U c\} \in Unify(U, U', v, c)}{Unify(U, U', v, c)}$$

$$\text{Unificação de Índices } \frac{for\, i \in \{0 \ldots n\}, \{\theta_i \mapsto_{typeof\, U'_i} \theta'_i\} \in Unify((U\,\theta), (U'\,\theta'), (v\,\theta), (c\,\theta'))}{Unify((U\,\theta), (U'\,\theta'), (v\,\theta), (c\,\theta'))}$$

$$\text{Inserção Simples } \frac{\{x \mapsto x'\} \in \Delta}{\{x \mapsto x'\} \in Unify(U, U', v, c) \qquad \{x' \mapsto y\} \notin \Delta}$$

$$\text{Anti-Círculo } \frac{\Delta, \{x \mapsto y\} \vdash \{x' \mapsto x\}}{\{x \mapsto x'\} \in Unify(U, U', v, c) \qquad \{x' \mapsto y\} \in \Delta}$$

$$\text{Substituição } \frac{\{t_\beta \mapsto t'_\beta\} \in \Delta}{\Gamma \vdash t = t'}$$

Regras de unificação no MPC

Cada regra acima corresponde a uma etapa que a unificação pode sofrer. Algumas vezes é possível notar que algumas regras parecem se sobrepor a outras. Nesse caso basta verificar se há alguma condição, por exemplo, ao inserir uma unificação em um contexto $Delta$. A regra *Inserção Simples* ocorre quando não existe dentro do contexto $Delta$ alguma unificação que leve x a alguma outra variável. Caso isso ocorra, a regra de *Anti-Círculo* deve ser aplicada. Isto é importante para garantir consistência em nosso contexto, uma vez que unificações mal aplicadas podem levar o programa a rodar infinitamente ou aceitar um programa inválido como correto.

A regra *Contexto Vazio* : esta regra indica que um contexto sem nenhuma definição de unificação dentro é válida. Isso indica haver programas que podem funcionar sem a necessidade de unificador.

A regra de *Introdução de substituição* : esta regra introduz uma unificação em um contexto Δ. Uma unificação é uma seta que aponta uma variável a (alvo) para b (objetivo), onde o tipo da unificação é o mesmo de b, indicado por U em $\{a \mapsto_U b\}$

A regra de *Anti-Reflexividade* : esta regra diz que para uma regra ser válida dentro de um contexto Δ, as duas variáveis unificadas não podem ser iguais. Em nosso caso igualdade diz respeito sobre a beta-convertibilidade.

A regra de *Anti-Simetria* : esta regra diz que para uma regra ser válida dentro de um contexto Δ, não pode existir a mesma unificação inversa no contexto Δ.

A regra de *Unificação Simples* : esta regra diz que as variáveis v e c são unificadas em $Unify(U, U', v, c)$, esse tipo de unificação ocorre quando o checador de cláusulas encontra um caso simples de análise de caso. Por exemplo, quando existe um variável x correspondendo aos números naturais em que x pode ser igual a 0 ou maior que 0, logo temos $\{x \mapsto_U 0\} \in Unify(Nat, Nat, x, 0)$, quando x é analisado como 0 e $\{x \mapsto_U (y+1)\} \in Unify(Nat, Nat, v, (y+1))$ quando x é maior que 0, para um número natural y qualquer. [2]

A regra de *Unificação de Índices* : esta regra é responsável por unificar índices de construtores, onde há presença de tipos dependentes. Por exemplo, em $Unify((Vector\, x, Vector\, 0, vec, vec'))$ o índice de x é 0, onde representa o tamanho do vetor unificado por $\{x \mapsto_U 0\}$. Isso garante que o tipo do vetor vec e vec' respeitam a igualdade por definição.

A regra de *Substituição* adiciona uma nova etapa adjunto com a β-redução e η-redução, termos que foram unificados podem ser livremente substituídos por outros termos, desde que respeitem a relações de $\rightarrow_\beta$ e $\rightarrow_\eta$.

5.4 Implementação de um verificador de tipos

O checador de tipos deste trabalho foi desenvolvido utilizando a linguagem de programação Haskell. Toda estrutura da linguagem é desenvolvida utilizando a seguinte estrutura algébrica:

```
1  data Term =
2     Pi Term Term Term
3    | Lam Term Term
4    | App Term Term
5    | Var VarName
6    | Constr Term [Term]
7    | Match Term Term [(Term, Term)]
8    | Notation Term Term deriving (Eq, Ord, Show)
9    -- Tactic [Term]
10
```

[2] Durante o desenvolvimento desse trabalho identificamos que nosso sistema de unificação é compatível com o axioma K, descrito nos trabalhos de Cockx (2017). Devido à regra de unificação simples, o axioma K implica em diversas mudanças no ponto de vista matemático, no entanto, essa discussão não está no escopo desse trabalho

Um termo de MPC, é algebricamente um termo Pi, uma expressão lambda, uma aplicação, uma variável, um construtor, um analisador de padrões e por fim uma notação de tipo.

```
data VarName = VarName Int | VarRef String deriving (Eq, Ord)
```

Uma variável é representada como uma estrutura que guarda ou o nome de uma variável usada principalmente para sinalizar o nome de uma função ou um número inteiro sinalizando uma variável criada por uma abstração lambda. Por exemplo, o termo $\lambda x.(funcx)$ é representado pela árvore $Lam\,(VarName\,0)\,(App\,(VarRef\,"func")\,(VarName\,0))$. O inteiro zero é gerado a partir de um gerador de nomes únicos. Isto evita conflitos entre variáveis inicializadas por termo lambda de terem conflitos. Esta técnica é alternativa à representação de Bruijn, onde os nomes dos termos são nomeados por um índice criado a partir da profundidade da posição das variáveis e das abstrações lambda (LESCANNE; ROUYER-DEGLI, 1995).

```
name :: String -> Unique Int
...

purefyPTerm :: PTerm -> Unique Term
purefyPTerm (PType term_name type_ body) = do
   term_name_v <- name term_name
   type__purified <- purefyPTerm type_
   body_purified <- purefyPTerm body
   type_branch <- substituteVarNames (term_name_v, term_name) body_purified
   return (Pi (Var . VarName $ term_name_v) type__purified type_branch)
purefyPTerm (PLam args body type_) = do
   bodyM <- purefyPTerm body
   x <- encodeAbstractions args bodyM
   case type_ of {
      Just type_ -> (do
         type__purified <- purefyPTerm type_
         return (Notation x type__purified));
      Nothing -> return x;
     }
purefyPTerm (PApp x y) = do
   purified_x <- purefyPTerm x
   purified_y <- purefyPTerm y
   return (App purified_x purified_y)
purefyPTerm (PVar s) =
   return . Var . VarRef $ s
purefyPTerm (PConstructors type_ constructors) = do
   type_M <- purefyPTerm type_
   constructors_M <- mapM purefyPTerm constructors
   return (Constr type_M constructors_M)
purefyPTerm (PMatch destructed type_ patterns) = do
   destructed_M <- purefyPTerm destructed
```

```
32    type_M <- purefyPTerm type_
33    patternsM <- mapM (\(predicate, body) -> do
34       pred <- purefyPTerm predicate
35       (predM, pred_vars) <- encodeMatchPredicate pred
36       bodyM <- purefyPTerm body
37       branchs <- foldr (\pair bodyM -> do
38          body <- bodyM
39          substituteVarNames pair body) (return bodyM) pred_vars
40       return (predM, branchs)
41       ) patterns
42    return (Match destructed_M type_M patternsM)
43 purefyPTerm (PNotation term type_) = do
44    termM <- purefyPTerm term
45    type_M <- purefyPTerm type_
46    return (Notation termM type_M)
47 purefyPTerm (PDef name body term) = do
48    termM <- purefyPTerm term
49    body <- purefyPTerm body
50    let subs = Var . VarRef $ name
51    return $ apply (\t -> if t == subs then body else t) termM
```

purefyPTerm transforma nossa árvore inicial em um termo onde todas as variáveis aparecem de maneira única. Desta forma, purificando os termos para que o nosso checador de tipos não precise se preocupar com variáveis com nomes duplicados.

Nosso checador de tipos captura este termo único e realiza uma análise nos diversos termos, a fim de verificar a validade do programa. No entanto, há diversas fases em que um processador de tipos dependentes necessita passar. O ciclo é uma união entre a validação de regras, normalização e unificação, onde não existe ordem pré-definida.

5.4.1 Validação de regras

Na etapa da validação de regras, todo termo é direcionado para uma checagem simples para regras definidas pelo sistema formal da nossa linguagem, por exemplo :

```
1 error
2   true + 0
```

De fato é bem óbvio que o resultado da adição de um número natural com um termo booleano tem resultado não definido. Portanto, nosso checador de tipos deve avisar o programador de possíveis problemas semânticos.

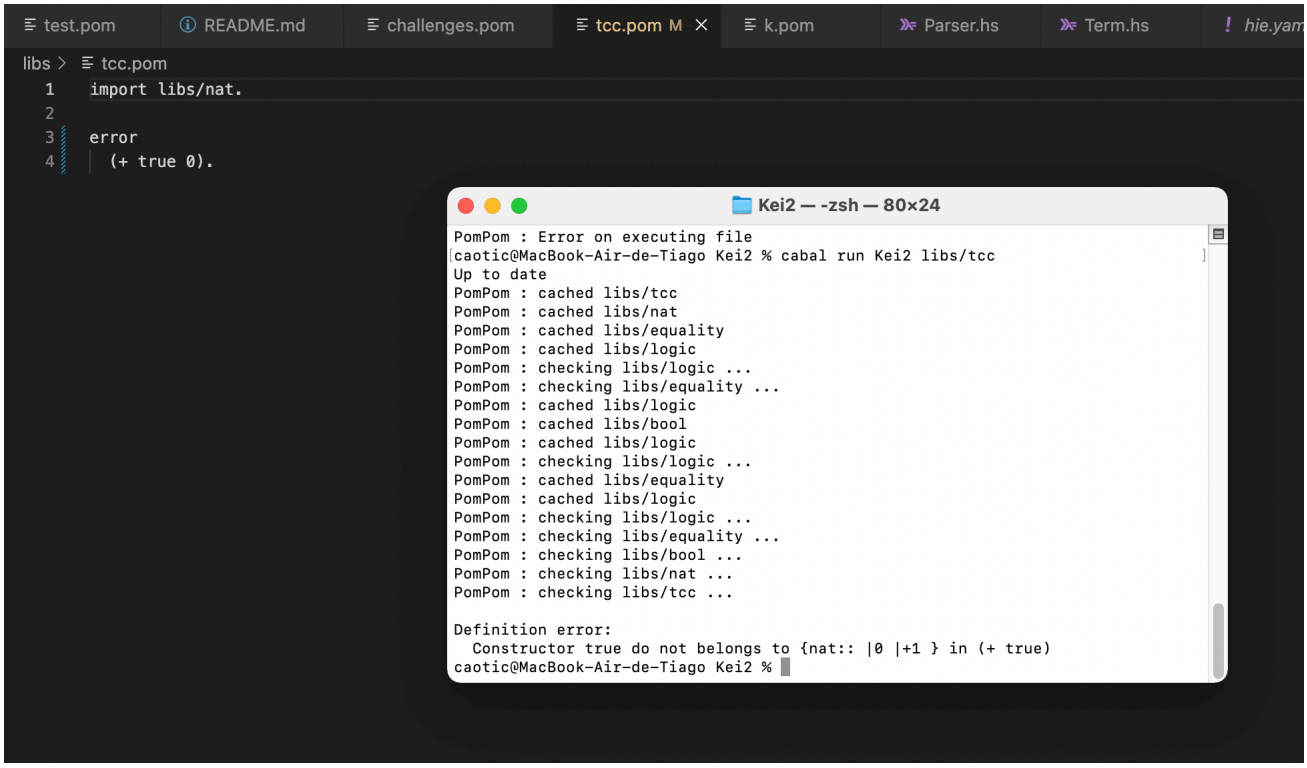

Figura 5 – Fonte : Próprio Autor

Nossa linguagem foi capaz de perceber o erro trivial cometido e avisou que a operação de adicionar não foi realizada corretamente, uma vez que o termo booleano verdade não faz parte do conjunto dos números naturais. Isso é apenas possível pois o assistente de provas conseguiu verificar na definição de soma o tipo $Nat \to Nat \to Nat$, onde Nat é representado pela hierarquia $\{nat :: |0| + 1\}$ (Lido como: conjunto de números naturais formado pelo construtor 0 ou algum sucessor).

Todas as regras simples checadas são derivadas de um super conjunto do cálculo lambda simplesmente tipado estendido com sub-tipagem, de tal forma que todo erro trivial será capturado pelo sistema de tipos. Nestas regras um contexto é usado para recordar sobre variáveis e seus tipos, por exemplo, no caso da função de adição.

$$ContextM := (setContextType, getContextType)$$

ContextM é a representação monádica de um contexto formado por duas operações principais, onde setContextType liga um termo a um tipo e getContextType retorna o tipo de um termo.

Decidir computacionalmente se um programa se encaixa nesse conjunto de regras costuma ser o mais trivial durante o processo de checagem. No entanto, os termos comumente podem representar computações, de tal forma que precisam ser computados antes de podermos aplicar tais regras.

```
1 error
```

```
2    ((refl Nat 4) :: (Eq (* 2 2) 4)
```

Nesse caso embora seja óbvio que 2 * 2 é igual a 4, a linguagem precisa computar o termo para afirmar a validade dele. Portanto é necessário passar por uma etapa de normalização.

5.4.2 Normalização

A normalização é a etapa onde termos que carregam algum potencial computacional são reduzidos para uma forma normal. Um termo está em sua forma β-normal se somente se não há presença de β-redex, isto é, se não existe um termo na expressão que possa ser reduzido. Por exemplo, $(\lambda\, x\,.\,x)\,(\lambda\, y\,.\,y)$ não está na sua forma normal porque contém um termo que pode ser reduzido para $(\lambda\, y\,.\,y)$ (LESCANNE; ROUYER-DEGLI, 1995).

```
1  normalize :: Term -> ContextM Term
2  normalize term = do
3          (check, norm_term) <- eagerStep term
4          if check then do
5             term_ <- showTerm norm_term
6             normalize norm_term
7          else
8             return norm_term
9    where
10     eagerStep :: Term -> ContextM (Bool, Term)
11     eagerStep t@(App(Lam x y) y') = do
12             (b, y) <- eagerStep y
13             (b', y') <- eagerStep y'
14             (b''', rec_) <- eagerStep (betaSubstitution (x, y') y)
15             return (True, rec_)
16     eagerStep t@(App (Notation term _) y') = eagerStep (App term y')
17     eagerStep t@(App x y) = do
18       (check, recur) <- do
19               (check_x, x_M) <- case x of {
20                 Var _ -> return (False, x);
21                 _ -> eagerStep x
22               }
23               y_M <- normalize y
24               return (check_x, App x_M y_M)
25       fun <- getFun t
26       case fun of {
27         Just v' -> (do
28           v' <- normalize v'
29           let definitional_term = reduceBetaVariables (expandFunction recur
     v')
30           let (matchs, resolvable_match) = checkMatchReduction
    definitional_term
31           if matchs == resolvable_match then do
```

```
32            return (True, definitional_term)
33          else
34            return (check, recur));
35          Nothing -> return (check, recur);
36        }
37    eagerStep v@(Var (VarRef name)) = do
38            term <- getRef name >>= mapM eagerStep
39            case term of {
40              Just k -> return (True, snd k);
41              Nothing -> return (False, v);
42            }
43    eagerStep m@(Match (Notation matched type_) type' k') = eagerStep $
    Match matched type' k'
44    eagerStep m@(Match matched type' k') = do
45            (checker, matchedM) <- eagerStep matched
46            (checker', type'M) <- eagerStep type'
47            let match = Match matchedM type'M k'
48            if checkIfUnstuck match then do
49               return (True, evalMatch match)
50             else
51               return (checker || checker', match)
52    eagerStep (Lam abs body) = do
53            (checker, bodyM) <- eagerStep body
54            return (checker, Lam abs bodyM)
55    eagerStep (Constr type_ constructors) = do
56            (checker, type_M) <- eagerStep type_
57            return (checker, Constr type_M constructors)
58    eagerStep (Pi var type_ body) = do
59            (checker, typeM) <- eagerStep type_
60            (checker', bodyM) <- eagerStep body
61            return (checker || checker', Pi var typeM bodyM)
62    eagerStep (Notation term type_) =
63            return (False, Notation term type_)
64    eagerStep v = return (False, v)
```

5.5 Resultados

A linguagem foi desenvolvida e distribuída de forma *opensource* não-licenciado[3], ela acompanha uma pequena biblioteca com definições básicas de dados e tipos disponíveis para construir programas e provas.

- Logic : Uma pequena coleção de definições para raciocinar sobre lógica de terceira ordem.

[3] *Opensource* : Código aberto, licença livre

- Bits : Uma pequena coleção de definições representando Bits
- Bool : Uma pequena coleção de definições representando Booleanos
- Nat : Uma pequena coleção de definições e teoremas do conjunto dos números naturais.
- Equality : Uma pequena coleção de definições e provas representando igualdade proposicional
- Function : Uma pequena coleção de definições para composição de funções.
- Functor e Monad : Uma pequena representação de conceitos relacionados a teoria de categorias para programação funcional.
- Vector : Uma pequena representação de um vetor

Para trabalhar com o MPC é necessário apenas criar um arquivo com a extensão MPC e chamar o interpretador pelo terminal como o caminho do arquivo, por exemplo MPC meu_arquivo_test.mpc.

5.5.1 Exemplos

Para importar bibliotecas é necessário usar a keyword import no começo de cada arquivo. Por exemplo, para usar a biblioteca sobre números naturais deve-se incluir dessa forma:

```
1 import libs/nat.
```

Como exemplificação, suponha-se que o usuário queira definir o conjunto de números pares e impares, vamos chamar esse arquivo então de paridade.mpc :

```
1 Static par : ~ Nat ~> *.
2 Static impar : ~ Nat ~> *.
3
4 Static par0 : (par 0).
5 Static par_suc : (x : Nat) ~ (impar x) ~> (par (+1 x)).
6 Static impar_suc : (x : Nat) ~ (par x) ~> (impar (+1 x)).
```

Iniciamos a estrutura par e impar como um tipo, como um conjunto dos números naturais para um conjunto de símbolos. Nesse caso par0, par_suc, e impar_suc,

Igualdade proposicional : <https://ncatlab.org/nlab/show/equality>

par0 define o caso básico, onde 0 é o primeiro número par, par_suc é o caso indutivo dos pares, onde o sucessor de um impar é um par, e finalmente impar_suc que define indutivamente que o sucessor de par é um impar.

```
even_half_mul_conserv
  |x :: (x : Nat)  ~> (Even (x * 2)) => __.
even_succ_pred
```

Caso o programador não tenha certeza do nosso objetivo, podemos deixar um buraco para que MPC informe a especificação do objetivo.

```
Definition even_half_mul_conserv:
 The hole expects (Even (x + x + 0))) type
```

Posteriormente podemos aplicar a analise de padrões para quebrar a prova em dois diferentes caminhos :

```
even_half_mul_conserv
  |x :: (x : Nat)  ~> (Even (x * 2)) => [x of (Even (x * 2))
    |0 => __
    |(+1 pred) => __
  ].
```

Onde x é igual a 0, o MPC nos diz que nosso objetivo é:

```
Definition even_half_mul_conserv:
 The hole expects a constructor of even0 or even_succ of type (even 0)
```

Podemos completar essa prova trivialmente, uma vez que 0 é um número par definido por nosso caso base pelo construtor even0.

```
even_half_mul_conserv
  |x :: (x : Nat)  ~> (Even (x * 2)) => [x of (Even (x * 2))
    |0 => even0
    |(+1 pred) => __
  ].
```

No caso em que x é maior que 0, o MPC informa o seguinte objetivo:

```
Definition even_half_mul_conserv:
 The hole expects a constructor of even0 or even_succ of type (even (pred *
    2 + 2)))
```

Nesse caso utilizamos o princípio da indução para obter $(Even(pred * 2))$, por definição o sucessor de um número par é impar, e posteriormente o sucessor de número impar é par, logo a prova se completa com:

```
even_half_mul_conserv
  |x :: (x : Nat)  ~> (Even (x * 2)) => [x of (Even (x * 2))
    |0 => even0
    |(+1 pred) =>
        def induction = (even_half_mul_conserv pred);
        (even_succ (+1 (mul pred 2)) (odd_succ (mul pred 2) induction))
  ].
```

Agora o MPC nos diz que checou o arquivo paridade.mpc com sucesso, logo nossa prova foi verificada formalmente pela nossa linguagem.

6 AUTOMATIZAÇÃO

Neste capítulo discutiremos diferentes abordagens para sínteses de programas a partir de algoritmos de busca informada. Primeiramente apresentaremos notações básicas formais para se referir aos objetos estudados, tais como o conceito de normalização e de reescrita. Em seguida apresentaremos sugestões de heurísticas e métodos para realizar buscas entre termos bem tipados. Por fim, concluiremos com exemplos e uma descrição de implementação dentro da nossa linguagem MPC.

6.1 Síntese de programas

Em termos formais, sínteses de programas, inferência de tipos e checagem de tipos são conceitos bastante intercaláveis. Dunfield e Krishnaswami (2020) argumenta que em um contexto Γ podemos determinar cada conceito apenas verificando se as meta-variáveis escolhidas são de entrada ou saída. Por exemplo, suponha-se um contexto $\Gamma \vdash x : T$. Podemos determinar cada conceito com o seguinte algoritmo:

```
1  PROCEDIMENTO checa_contexto(Γ : Contexto, x : Termo, T : Tipo)
2    se (x != nulo) e (T != nulo) então
3      cheque se o tipo de x é igual a T no contexto Γ
4    fim se
5    se (x != nulo) e (T == nulo) então
6      infira o tipo de x no contexto Γ
7    fim se
8    se (x == nulo) e (T != nulo) então
9      sintetize um termo x que tem o tipo de T
10   fim se
11 FIM PROCEDIMENTO
```

Nem sempre é possível descrever um algoritmo para cada conceito apresentado, de forma simplificada a complexidade de cada problema aumenta a partir da quantidade de omissão de informações. Por exemplo, é provável que seja mais fácil de construir um algoritmo capaz de decidir quando dois termos são iguais (checagem de tipo) do que construir para inferir um tipo a partir de um termo (inferência de tipos), e por consequência sintetizar um termo a partir de um tipo é uma tarefa bastante difícil.

É importante destacar que diferentemente dos conceitos de checagem e inferência de tipos, a síntese de programas não é um conceito inerentemente ligado com a teoria de tipos. Por exemplo, Gulwani, Polozov e Singh (2017) define a síntese de programas como uma tarefa automática de gerar um programa a partir de um conjunto de restrições expressadas em algum sistema lógico. Convenientemente é possível pensar que as relações que os tipos expressam é a de um cálculo de restrições. No entanto, seria incorreto limitar esse pensamento apenas à teoria de tipos, uma vez que existem outros sistemas de restrições tais como a lógica Hoare e a programação lógica.

6.2 Reconstrução de provas

Durante o processo de automação é frequente a utilização de ferramentas terceiras como verificadores de modelos e resolvedores SAT para auxiliar no processo de automatização (PAULSON; SUSANTO, 2007). No entanto, cada ferramenta implementa uma maneira diferente de resolução. Por exemplo, Paulson e Blanchette (2015) e Czajka e Kaliszyk (2018) empregam representações de primeira ordem para converter termos das linguagens a um formato que cada automatizador suporta, enquanto Foster e Struth (2011) utilizam-se de uma representação puramente equacional.

Enquanto a utilização de ferramentas garante uma vasta quantidade de possibilidades, uma vez que o usuário pode escolher qual ferramenta se comporta melhor ao provador, ela também exige a necessidade de reconstrução da prova entre os dois lados, o do provador e o da ferramenta. Primeiro o provador precisa especificar o objetivo da prova na linguagem da ferramenta, de modo que a ferramenta possa devolver o objetivo para poder converter novamente o objetivo em um formato que a linguagem possa validar (PAULSON; SUSANTO, 2007). É relevante ressaltar que a ferramenta pode falhar na obtenção do objetivo. Além disso, Paulson e Susanto (2007) ainda ressalva que mesmo que a prova seja conquistada, ela pode não ser válida nos termos da linguagem alvo.

6.3 Planejadores

Sistemas de planejamentos descrevem sequências lógicas de etapas para a resolução de um objetivo a partir de um estado inicial, onde filtramos um conjunto de ações que potencialmente levariam ao estado objetivo (ALVES, 2014). Alves (2014) e Tonidandel e Rillo (2002) descrevem formalmente um sistema de planejamento como uma tripla (A, I, G), onde A é conjunto de etapas que o sistema pode sofrer, I é o estado inicial e G o estado objetivo. Uma ação c é formada por uma tripla $(pre(c), add(c), del(a))$, onde cada elemento representa uma mudança de estado das preposições quando a ação c é executada, por exemplo, $pre(c)$ representa as preposições que o sistema precisa obedecer para c ser executado. Já $add(c)$ significa as ações que serão verdadeiras a partir do momento em que c é executada. Por fim, $del(c)$ representa as preposições que deixaram de ser verdadeiras após c ser executada.

É fácil observar a relação entre sistemas de planejamento e automatizadores de provas. Por exemplo, Buch e Hillenbrand (1998) cita como exemplo o que seria um possível design de um provador:

```
1  Enquanto o problema não se resolve, faça:
2    (a) Selecione uma regra de inferência R;
3    (b) Selecione uma tupla de fatos x;
4    (c) Aplique uma regra de inferência R a uma tupla de fatos x.
```

Fonte: Buch e Hillenbrand (1998)

Informalmente podemos dizer que uma ação c de um sistema de planejamento é o algoritmo descrito acima, a $pre(c)$ significaria o conjunto de regras, inferências que precisamos aplicar, $add(c)$ os fatos selecionados da tupla, e por fim, $del(a)$ os fatos que deixaram de pertencer ao novo estado.

6.3.1 Automação de teorias equacionais

Os trabalhos de KNUTH e BENDIX (1970) introduzem um novo tipo de cálculo capaz de generalizar problemas algébricos através de um modelo equacional. Nele é informalmente representado por dois lados de uma equação $\alpha = \beta$, onde um lado é a forma simplificada de outro lado. Por exemplo, α é simplificado como β, este modelo se tornou conveniente para a automatização de provas como no caso de Buch e Hillenbrand (1998), onde foi proposto o automatizador de prova WALDMEISTER, baseada em saturação. WALDMEISTER realiza uma série de reescritas a partir de um conjunto de regras e equações. Outras estratégias baseadas em heurísticas também são aplicadas quando há presença de pares críticos, isto quando há duas equações que conflitam entre si, possibilitando uma taxa de sucesso de acordo com cada estratégia heurística empregada (BUCH; HILLENBRAND, 1998).

Por se tratar de uma algébrica generalizada, a automação por teorias equacionais pode ser aplicada em outros provadores, como no caso do trabalho de Jones (2019) com base na linguagem de tipos dependentes Agda. Naquele trabalho cada equação é convertida para ser uma sequência de etapas que respeitam as regras de inferências da linguagem.

6.4 Exemplos de expansão de nós

Para realizar uma prova é necessário utilizar-se do conhecimento anteriormente construído. Este pode ser trivialmente uma regra de inferência, um teorema ou um conjunto de ambos. Portanto, durante uma busca para um objetivo X precisamos ser capazes de gerar expansões desta codificação a fim de resolver alguma meta (ALVES, 2014).

Durante o processo de automação de prova é comum que sistemas de automatização realizem uma busca entre as definições já presentes na linguagem. Alguns exemplos são os sistemas Hammers que utilizam de força bruta e algoritmos de machine learning para derivar informações previamente definidas (CZAJKA; KALISZYK, 2018) e sistemas de filtragem de relevância que realizam a seleção de definições mais significantes para o objetivo final (MENG; PAULSON, 2009). Embora ambos os métodos consigam expandir informações, a filtragem tem absoluto impacto em cima de sistemas de automatização por busca informada. Meng e Paulson (2009) constataram que um bom critério para a filtragem é a capacidade de aumentar o sucesso de teoremas provados além de economizar recursos computacionais, uma vez que o espaço de busca para a realização dos procedimentos é significativamente reduzido.

6.4.1 *Expansão por algoritmo evolucionário*

Processos estocásticos não-determinantes apresentam um relativo sucesso para a expansão e seleção de nós (YANG *et al.*, 2016). Os autores identificaram a relação entre a procura de uma prova com a síntese de um programa, onde neste último existe uma variedade de estudos usando Programação Genética.

```
1  PROCEDIMENTO SelecionaUmPai(Pop)
2    i <- UniformInt(0, Pop.size/2)
3    retorne o indivíduo i-th com maior fitness
4  FIM PROCEDIMENTO
```

A partir de um *sample* (população) um novo pai é escolhido por um parâmetro de fitness. No trabalho de YANG *et al.* o fitness foi atribuído pela quantidade de táticas utilizada com a condição de que cada tática atendesse.

6.4.2 *Filtro de relevância*

Mesmo em condições onde um provador de teoremas automático consegue provar qualquer prova na condição em que esta seja verdadeira, ainda existem diversos problemas que cercam a automação de provas. O maior problema é a falta de recursos computacionais para achar uma prova. Por exemplo, o caso em que contém um conjunto de axiomas bem específico. No entanto, a quantidade de axiomas irrelevantes é grande. Paulson e Blanchette (2015) citam esse problema e reafirmam o potencial de filtros de relevância no aumento da taxa de sucesso em relação à automação de provas.

Um filtro de relevância pode, por exemplo, filtrar axiomas que são pouco referenciados, a fim de resguardar axiomas-chave para a maioria dos objetivos (MENG; PAULSON, 2009). Este processo não só possibilita melhor velocidade na busca, mas também permite aumentar a taxa de sucesso, caso a procura seja baseada em algum tipo de saturação.

6.5 Notações

Em nosso trabalho o conceito de sintetização de programas será extensivamente estudado em virtude da automatização de provas, uma vez que ambos conceitos estão intimamente associados pela correspondência entre programas e provas (3.3.3).

Um problema de automatização de provas em MPC é um conjunto de premissas H_n, onde n está contida em um intervalo de $1...k$ dentro dos números naturais. Um objetivo final O, por fim, é necessário através de uma sequência de passos finitos. Visamos então construir um termo bem tipado em que O é um objeto habitado.

```
1  H1 : Nat
2  H2 : (Even H1)
3  ----------------------
4  O : (2 * (half H1)) = H1
```

Cada premissa corresponde um nome associado a um termo do MPC. O objetivo O é o resultado do retorno da definição, isto é, o objetivo da definição:

```
|H1 H2 :: (H1 : Nat) ~> ~ (Even H1) ~> (Eq Nat H1 (mul 2 (half H1))) => O.
```

6.5.1 *Vértices para expansão*

Supondo um problema de satisfação de restrição de um ponto A ao B, onde A é o estado inicial e B é o estado final descrito como o objetivo, podemos representar cada etapa por um grafo, onde seu nó é um estado qualquer e cada vértice é uma transição de estado, representado como:

$$\rightarrow_{\beta} : \text{Normalização}$$
$$\rightarrow_{APP_f} : \text{Aplicação}$$
$$\rightarrow_{CASEc} : \text{Análise de caso}$$

É notório que nem toda transição leva a uma relação de um estado para outro estado, por exemplo, em $\rightarrow_{CASEc}$, onde c é um construtor, para cada cláusula c' de c um novo estado será gerado.

$$\rightarrow_{CASEn} \rightarrow_{APP_f}$$

6.6 MiniLambda : Um protótipo de automatizador de alta ordem

Com intenção de resolver um problema de automatização de prova, MiniLambda é dotado de um algoritmo multi-agente, de forma a quebrar o problema em menores partes e arquitetar uma busca coletiva.

6.6.1 *Representação de Termos*

Durante o trabalho de automatização de provas é comum utilizar representações de diferentes ordens como discutido acima. No entanto, não necessariamente ambas representações precisam divergir de ordem. Por exemplo, um automatizador X poderia usufruir de uma representação Y, de tal maneira que X e Y mantenham a mesma ordem. Isto é conveniente porque a conversão dos termos se torna ainda mais direta. Neste trabalho propomos uma representação de alta ordem generalista para a automação de provas no contexto do MPC, no entanto, não limitado apenas a este.

A representação de um termo em MiniLambda é semelhante aos termos empregados em MPC. Podemos dizer que um termo em MiniLambda é representado pelo tipo de dados abaixo:

```
data VarRef = Name String | Paralell Int
data MiniLambda =
    App MiniLambda MiniLambda
    |Abs String MiniLambda
    |Var VarRef
    |Pi String MiniLambda MiniLambda
```

Um termo em MiniLambda é representado acima como um cálculo simples de alta ordem, representado por uma tupla constituída de uma aplicação de funções, uma abstração—lambda, uma variável e finalmente um tipo. Embora a semelhança com os termos de MPC, há uma necessidade de conversão dos termos. Isto é importante para garantir a validade dos termos. Paulson e Susanto (2007) pontuam sobre a necessidade dessa validação devido a dois problemas. Primeiro pela possibilidade de haver erros introduzidos na interface do código, traduzindo para um problema do automatizador. Segundo pela chance de algum erro ocorrer durante a transformação do termo do automatizador para a linguagem alvo, em nosso caso de MiniLambda para o MPC.

Definimos uma relação $MiniLambda \leftrightarrow MPC$ chamada de R, onde f é uma função sobrejetora $MiniLambda \rightarrow MPC$ que transforma um termo de MiniLambda para MPC e g é uma função parcial $MPC \rightarrow MiniLambda$ que transforma um termo de MPC para MiniLambda. R deve satisfazer $g\,(f\,x) = x$ e $f\,(g\,y) = y$, se $f\,(g\,y) \in MiniLambda$. É importante a ressalva que embora R possa ser um isomorfismo a depender da linguagem alvo, isso não é regra para todas as linguagens. Por exemplo, em nossa relação entre MPC e MiniLambda, R não é um isomorfismo.

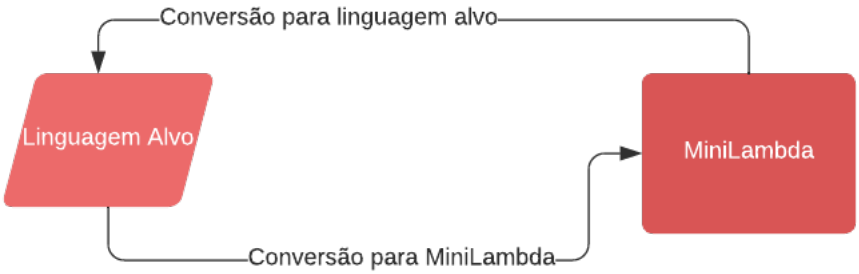

Figura 6 – Fonte: Próprio Autor

6.6.2 Representação de um problema

A representação de um problema em MiniLambda é dada por um contexto C, um objetivo O, onde C é uma lista de cláusulas, onde cada cláusula é formada por uma dupla (N, T), em que N é um nome de uma variável disposta no contexto C e T é um tipo dessa variável representado por um termo de MiniLambda, sendo que O é um termo MiniLambda correspondente pelo tipo do objetivo do problema.

```
1 h = A
2 f = [(x : A) -> B]
3 f2 = [(y : A) -> C]
4 : B
```

No exemplo acima o contexto é formado por $\{h, f, f2\}$ e o objetivo é B. Um problema é considerado solucionado se MiniLambda sintetizar um termo T no contexto do C, tal que $C \vdash T : B$.

6.6.3 Arquitetura do agente

Cada agente está situado em um espaço, mais especificamente podemos descrever este espaço como um *problem-space graph*, onde cada nó representa um estado do problema e seus vértices são as operações entre os agentes (KORF, 1996).

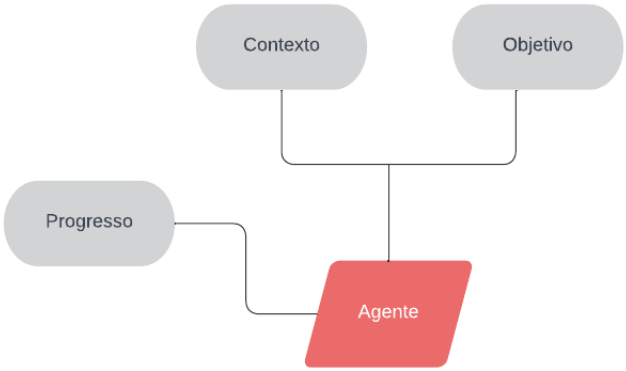

Figura 7 – Fonte: Próprio Autor

Estendemos um problema com P. P é uma estrutura chamada de progresso com o papel de abstrair a busca de um problema. P é uma tupla, cujo valor pode ser nulo, onde em (P_1, P_2), P_1 é um termo de tipo P_2, ou seja, $C \vdash P_1 : P_2$. Posteriormente, podemos representar esta estrutura como uma instância de um problema, onde um contexto C é nosso ambiente, P nosso estado mutável e O nosso objetivo final (KORF, 1996).

```
1 PROCEDIMENTO criar_agente(T : Contexto, O : Objetivo) : Agente
2    retorne No(T, O, Nulo)
3 FIM PROCEDIMENTO
```

Variáveis nem sempre denotam apenas valores comuns, alternativamente variáveis podem referenciar outros termos paralelamente. Isto é interessante para representar agentes dentro de termos de MiniLambda. Denotamos $Var\,(Parallel\,N)$, onde N é um nome de um agente quando um termo referir-se a um agente. Um nome de um agente é a forma em que algumas vezes o sistema se refere a ele. Isso é importante porque agentes com o mesmo nome representam o mesmo espaço de problema.

$$HASH(NO_{ATUAL}) = HASH(NO_{PAI}) + HASH(C) + HASH(O),$$

Onde C e O denotam o contexto e o objetivo respectivamente.

Um agente pode depender de outro agente, dessa forma chamamos $HASH(NO_{ATUAL})$ como filho do NO_{pai} por um processo de expansão de nós (KORF, 1996). Chamamos de nó de gênesis um espaço de um problema em que não há presença de um NO_{pai}.

Agentes são endereçados em uma estrutura global, onde agentes com o mesmo nome compartilham do mesmo endereço em um formato de lista, portanto agentes com o mesmo nome podem coexistir na mesma árvore de busca.

6.6.4 Funções de exploração

Um agente em MiniLambda contém uma lista de operações em que uma ação é escolhida a partir da inferência entre contexto e progresso. Essas ações geram um novo estado para um progresso e podem criar novos agentes exploradores.

Quando o progresso de um agente é nulo, aplicamos uma função básica de inicialização do progresso. Esta função busca no meio do contexto através de uma simples heurística um termo para se encaixar como um progresso inicial.

Descrevemos a heurística como um simples algoritmo de comparação simbólica através de uma relação de alfa-equivalência $=_\alpha$:

$$
\begin{aligned}
&filter(x,y) = True, se\ x =_\alpha y \\
&filter(x,y) = True, se\ last(x) =_\alpha y \\
&filter(x,y) = False, do\,contrario \\
&onde \\
&\quad last(Pi\,name\,type\,body) = last(body) \\
&\quad last(x) = x, do\,contrario
\end{aligned}
$$

$filter$ é aplicada no contexto C para filtrar cláusulas para qualquer dupla (N,T) em que $filter(O,T)$ seja verdade, onde O denota o objetivo do agente. A partir das cláusulas filtradas, o algoritmo sorteia uma única cláusula. Caso não exista nenhuma cláusula que seja compatível com a heurística, uma cláusula qualquer é escolhida.

```
1  PROCEDIMENTO tentar_primeiro_termo(agent : No) : agent
2      contexto <- getContexto agent
3      objetivo <- getObjetivo agent
4      clausulas_filtradas <- filtrar_clausulas(contexto, objetivo)
5      pertubação <- clausulas_filtradas[Math.random()]
6      se pertubação == nulo:
7         -- seleciona qualquer clausula aleatoria
8         clausula_aleatoria <- contexto[Math.random()]
9         retorne clausula_aleatoria
10     retorne pertubação
11 FIM PROCEDIMENTO
```

Se um termo precisa ser desenvolvido, por exemplo, na situação onde o termo é uma função e necessita ser expandido através de seus argumentos, uma função de exploração com um algoritmo de expansão de nó é empregada. Por exemplo, quando, dado agente, onde seu progresso é denotado pela função $f : A -> B$ sobre um contexto C, f é expandida sobre o argumento de A, logo o novo agente é a tripla $(C, B, nulo)$.

```
PROCEDIMENTO aplique_pi(agent : No) : agent
    (P1, P2) <- getProgresso agent
    contexto <- getContexto agent
    retorne (Var (Parallel (criar_agentes(contexto, head(P2)))))
FIM PROCEDIMENTO
```

Onde a função head é demonstrada por:

$$head(Pi\, name\, type\, body) = type$$
$$head(x) = erro, do\, contrario$$

A função $aplique\_pi$ cria uma dependência entre dois agentes ou mais, na presença de um tipo Pi denotado por $\Pi x : P(x)$, onde x aparece no termo dependente. A função realiza a aplicação da função substituindo os termos por referência dos agentes.

$$subs(\Pi x : P(x)) = P(x)[x/agent\_id],$$

Onde $agent\_id$ é o id dos novos agentes criados

A expansão de nó também é empregada quando há presença de um tipo Pi. Isto ocorre pois o progresso é uma abstração lambda:

```
PROCEDIMENTO criar_beta_expansão(agent : No) : agent
    (termo, tipo) <- getProgresso agent
    context <- getContext agent
    nova_variavel <- string(Math.random())
    retorne (Abs nova_variavel criar_agent(expandir_contexto(context,
   nova_variavel, head(tipo)), last(tipo)))
FIM ALGORITMO
```

$expandir\_contexto$ é uma função responsável por anexar uma nova variável em um contexto, portanto temos que $(N, T) \in expandir\_contexto(C, N, T)$ e $C \subset expandir\_contexto(C, N, T)$.

Finalmente, descrevemos funções de explorações puramente baseadas nas operações do cálculo λ. Estas operações ocorrem sempre que possível e têm como papel simplificar os termos através do conjunto de operações normalizadoras [1]:

$$aplique\_beta(termo) = \rightarrow_\beta$$
$$aplique\_eta(termo) = \rightarrow_\eta$$

[1] $aplique\_beta$ e $aplique\_eta$ é o conjunto de operações descrito no capítulo 3.2

Podemos especificar a função explorar como uma sequência destas operações. Esta função retorna o estado do progresso a cada iteração onde o algoritmo é incrementado.

```
PROCEDIMENTO explorar(agent : No) : No
   se o objetivo foi alcançado então
       retorne agent
   termo <- getProgresso(agent)
   se termo == nulo então
      retorne tentar_primeiro_termo(agent)
   se termo é eta-redutivél:
      retorne aplique_eta(agente)
   se termo é beta-redutivél:
      retorne aplique_beta(agente)
   se termo é pi-redutivél:
      retorne aplique_pi(agente)
   se o termo é uma abstração lambda:
      retorne beta_expansão(agente)
   se termo é mal tipado:
       retorne nulo
FIM PROCEDIMENTO
```

6.6.5 Algoritmo Multi-Agente

Todos os agentes dispostos no ambiente têm capacidade de acessar informações de outros agentes. Isso é relevante porque termos frequentemente dependem de outros termos. Para isso existe uma função que realiza o acesso aos termos que dependem recursivamente:

```
isEqualParalellTerm :: TreeType -> TreeType -> Search Bool
isEqualParalellTerm (Var (Paralell k)) (Var (Paralell k')) = do
    (Problem (SearchAST goal _) _) <- getNode k
    (Problem (SearchAST goal' _) _) <- getNode k'
    isEqualType goal goal'
isEqualParalellTerm _ _ = return False

isEqualType :: TreeType -> TreeType -> Search Bool
isEqualType p@(Var (Paralell k)) term' = do
    term <- getTerm <$> getNode k
    case term of {
        Just (expr, term) -> isEqualType expr term';
        Nothing -> isEqualParalellTerm p term'
    }
isEqualType term' p@(Var (Paralell k)) = do
    term <- getTerm <$> getNode k
    case term of {
        Just (expr, term) -> isEqualType term' expr;
        Nothing -> isEqualParalellTerm p term'
    }
```

```
isEqualType (App x y) (App x' y') = do
    b_x <- isEqualType x x'
    b_y <- isEqualType y y'
    return (b_x && b_y)
isEqualType (Abs name body) (Abs name' body') = do
    body <- (substitute (Var $ Name name, Var $ Name name') body)
    isEqualType body body'
isEqualType v v' = return $ v == v'
```

isEqualType pode ser descrita como função que compara o progresso (termo) de dois agentes e a referência do seus dependentes. Se uma referência tem o mesmo nome para o mesmo agente, utilizamos o primeiro agente com o termo válido. Caso falhe, podemos utilizar outra combinação de agentes dependentes.

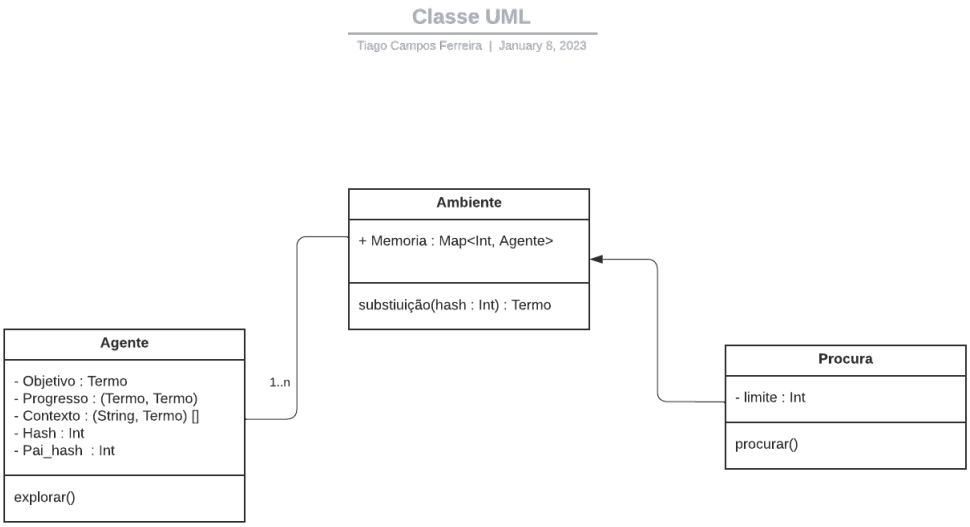

Figura 8 – Fonte: Próprio Autor

O algoritmo proposto realiza uma busca via saturação, definimos um limite até onde o algoritmo pode buscar uma solução válida.

```
PROCEDIMENTO procurar(agents : Map<No>, max : inteiro) : List<No>
   se max == limite então
      retorne agents
   para todo agent em agents:
      explorar(agent)
   retorne procurar(agents, max+1)
FIM PROCEDIMENTO
```

Um termo é considerado válido se o agente gênesis tem o progresso correspondente ao tipo do objetivo. Além disso, toda e qualquer dependência do agente gênesis deve ser válida. Checamos a validade de um agente recursivamente pela função escrita abaixo:

```
constructLambdaTerm :: Int -> WrapMemo Search MiniLambda
constructLambdaTerm k = do
    node <- liftM' $ getNode k
    r <- remember k
    case (getTerm node) of {
        Just (term, type_) -> (do
               term <- readTerm term
               memoNode k term
               return term
           );
        Nothing -> return $ Var (Name "?")
    }
    where
        readTerm (App x y) = do
            x <- readTerm x
            y <- readTerm y
            return $ App x y
        readTerm (Abs name body) = do
            body <- readTerm body
            return $ Abs name body
        readTerm (Pi name type_ body) = do
            type_ <- readTerm type_
            body <- readTerm body
            return $ Pi name type_ body
        readTerm v@(Var (Name m)) = return v
        readTerm (Var (Paralell p)) = do
            r <- remember p
            case r of {
                Just term -> return term;
                Nothing -> constructLambdaTerm p;
            }
```

Caso o termo retorne qualquer variável “?”, o algoritmo falhou em resultar um termo válido. Se uma referência tem o mesmo nome para o mesmo agente, utilizamos o primeiro agente com o termo válido.

6.6.6 Resultados

MiniLambda foi desenvolvida e distribuída com código livre não-licenciado. Os testes foram realizados utilizando preposições extraídas da biblioteca da assistente de prova MPC. Utilizamos o parâmetro de 200 iterações como limite.

```
trivial_proof
 | sort x y eq pa ::
                      (Sort : *)
                      (x : sort)
                      (y : sort)
                      ~ (Eq H x y) ~>
                      ~ (proofAtoB x) ~>
```

```
8                          (proofAtoB y) => proof.
```

$trivial_proof$ Tem sua representação em MiniLambda traduzida para:

```
1 eqind = [(P : [(v : A) -> B]) -> [(we : ((Eq A) B)) -> [(wu : (P A)) -> (P
     B)]]]
2 pa = (proofAtoB A)
3 eq = ((Eq A) B)
4 : (proofAtoB B)
```

No contexto foi adicionado a função $eqind$ incluída nas bibliotecas de igualdade do assistente de prova MPC.

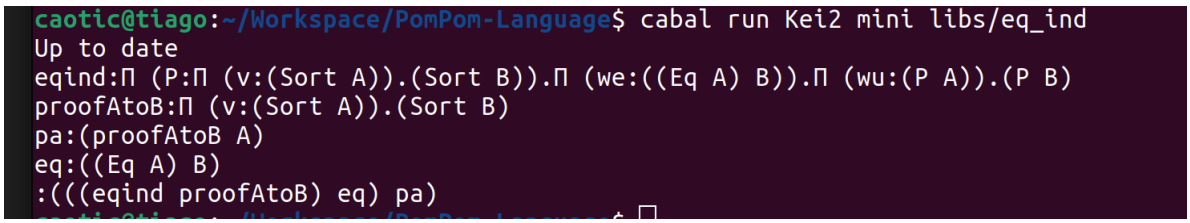

Figura 9 – Fonte: Próprio Autor

Descrevemos os passos em que MiniLambda procede através dos conjuntos de operações dos agentes. Ilustramos com a seta ”–>“ para demarcar quando um agente necessita de outro para realizar uma operação, por questão de legibilidade representamos apenas um agente por novo nó:

Agente Gênesis --> eq_ind
- Agente Y --> (v : A) $\lambda v.?$
  - Agente Y --> Agente Y' --> proofToB ?
    - Agente Y --> Agente Y'' --> v
- aplique_eta --> $\lambda v.(proofToB\, v) \rightarrow_\eta proofToB$
- Agente Y --> Agente X : eq
- Agente Y --> Agente X' : pa

Finalmente, podemos representar a automatização da prova em MPC, e validamos a automatização aplicando no assistente de prova:

```
1 trivial_proof
2  | sort x y eq pa ::
3                        (Sort : *)
4                        (x : sort)
5                        (y : sort)
```

```
                          ~ (Eq H x y) ~>
                          ~ (proofAtoB x) ~>
                            (proofAtoB y) => (((indind proofAtoB) eq) pa).
```

7 CONCLUSÃO

Durante a introdução desse trabalho sintetizamos conceitos-chave para introdução de técnicas formais para a formalização de provas baseada em computadores, principalmente no âmbito de assistentes de provas, onde apresentamos o assistente de prova MPC com o núcleo baseado no cálculo de $\lambda\Pi$, justificamos suas regras de inferência pela simplicidade e expressividade alcançada através do sistema de unificação descrito no trabalho.

No desenvolvimento desse trabalho abordamos questão relevantes a usabilidade de assistentes de provas, notadamente o problema de resolução de provas triviais, argumentamos que este problema reduz a produtividade de formalizadores, ou mesmo inviabiliza a formalização de algumas provas em um assistente de prova. Seguidamente, discutimos a abordagem de automatização de provas para a resolução de provas triviais, não obstante, defendemos o uso da inteligência artificial, em destaque métodos baseados em sistemas multi-agente para a resolução de provas.

No último capítulo descrevemos a ferramenta de automatização de prova MiniLambda, em que aplicamos um algoritmo multi-agente para resolução de provas do assistente de provas MPC. Concluímos com um exemplo de automatização de prova trivial no assistente de provas MPC utilizando o MiniLambda como ferramenta. Como trabalhos futuros destacamos a necessidade de medir a acurácia do MiniLambda frente a outros automatizadores disponíveis na literatura, de forma a comparar os parâmetros de sucesso, além disso, é interessante analisar outros meios de heurísticas com o intuito de melhorar o desempenho da automatização.

REFERÊNCIAS

ABEL, A. An introduction to dependent types and agda. 2009.

ALVES, R. M. F. Planejamento baseado em busca e aprendizagem. 2014.

BANDARU, S.; DEB, K. Metaheuristic techniques. In: . [S.l.: s.n.], 2016.

BLANCHETTE, J.; KALISZYK, C.; PAULSON, L.; URBAN, J. Hammering towards qed. **Journal of Formalized Reasoning**, Alma Mater Studiorum (Bologna), v. 9, n. 1, p. 101–148, 2016. ISSN 1972-5787.

BRAILSFORD, S. C.; POTTS, C. N.; SMITH, B. M. Constraint satisfaction problems: Algorithms and applications. **European Journal of Operational Research**, v. 119, n. 3, p. 557–581, 1999. ISSN 0377-2217. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0377221798003646>.

BUCH, A.; HILLENBRAND, T. Waldmeister: Development of a high performance completion–based theorem prover. 06 1998.

CHANDRA, S.; GORDON, C. S.; JEANNIN, J.-B.; SCHLESINGER, C.; SRIDHARAN, M.; TIP, F.; CHOI, Y. Type inference for static compilation of javascript. In: . New York, NY, USA: Association for Computing Machinery, 2016. (OOPSLA 2016), p. 410–429. ISBN 9781450344449. Disponível em: <https://doi.org/10.1145/2983990.2984017>.

COCKX, J. **Dependent Pattern Matching and Proof-Relevant Unification**. Tese (Doutorado), 06 2017.

COUSINEAU, D.; DOWEK, G. Embedding pure type systems in the lambda-pi-calculus modulo. In: . [S.l.: s.n.], 2007. p. 102–117. ISBN 978-3-540-73227-3.

CZAJKA, I.; KALISZYK, C. Hammer for coq: Automation for dependent type theory. **Journal of Automated Reasoning**, v. 61, 06 2018.

DEEPA; SENTHILKUMAR. Swarm intelligence from natural to artificial systems: Ant colony optimization. **International Journal on Applications of Graph Theory In wireless Ad Hoc Networks And sensor Networks**, v. 8, p. 9–17, 03 2016.

DELAHAYE, D. A tactic language for the system coq. In: . [S.l.: s.n.], 2000. p. 85–95. ISBN 978-3-540-41285-4.

DIXON, L.; FLEURIOT, J. A proof-centric approach to mathematical assistants. **J. Applied Logic**, v. 4, p. 505–532, 12 2006.

DUNFIELD, J.; KRISHNASWAMI, N. **Bidirectional Typing**. 2020.

DURFEE, E.; ROSENSCHEIN, J. Distributed problem solving and multi-agent systems: Comparisons and examples. 08 1995.

DéHARBE, D.; MOREIRA, A.; RIBEIRO, L.; RODRIGUES, V. Introdução a métodos formais: Especificação, semântica e verificação de sistemas concorrentes. **RITA**, v. 7, p. 7–48, 01 2000.

FERBER, J.; GUTKNECHT, O.; MICHEL, F. Agent/group/roles: Simulating with organizations. 04 2008.

FOSTER, S.; STRUTH, G. Integrating an automated theorem prover into agda. In: . [S.l.: s.n.], 2011. p. 116–130. ISBN 978-3-642-20397-8.

FRIPERTINGER, H.; SCHöPF, P. Endofunctions of given cycle type. **Annales des Sciences Mathématiques du Québec**, v. 23, p. 173–187, 01 1999.

GEUVERS, J. Proof assistants : history, ideas and future. **Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)**, Springer, v. 34, n. 1, p. 3–25, 2009. ISSN 0256-2499.

GULWANI, S.; POLOZOV, A.; SINGH, R. **Program Synthesis**. NOW, 2017. v. 4. 1-119 p. Disponível em: <https://www.microsoft.com/en-us/research/publication/program-synthesis/>.

HINDLEY, J.; SELDIN, J. **Introduction to Combinators and Lambda-Calculus.** [S.l.: s.n.], 1986.

JAFFAR, J.; LASSEZ, J.-L. Constraint logic programming. In: **Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1987. (POPL '87), p. 111–119. ISBN 0897912152. Disponível em: <https://doi.org/10.1145/41625.41635>.

JIM, T.; PALSBERG, J. **Type Inference in Systems of Recursive Types With Subtyping**. 1999.

JONES, S. P. **Type inference as constraint solving: how GHC's type inference engine actually works**. 2019. Zurihac keynote talk. Disponível em: <https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/>.

KNUTH, D. E.; BENDIX, P. B. Simple word problems in universal algebras††the work reported in this paper was supported in part by the u.s. office of naval research. In: LEECH, J. (Ed.). **Computational Problems in Abstract Algebra**. Pergamon, 1970. p. 263–297. ISBN 978-0-08-012975-4. Disponível em: <https://www.sciencedirect.com/science/article/pii/B978008012975450028X>.

KORF, R. E. Artificial intelligence search algorithms. In: **In Algorithms and Theory of Computation Handbook**. [S.l.]: CRC Press, 1996.

LANGLEY, P.; LAIRD, J. Artificial intelligence and intelligent systems. 01 2006.

LEROY, X. Formal verification of a realistic compiler. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 7, p. 107–115, jul 2009. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1538788.1538814>.

LESCANNE, P.; ROUYER-DEGLI, J. Explicit substitutions with de bruijn's levels. In: . [S.l.: s.n.], 1995. v. 914, p. 294–308. ISBN 978-3-540-59200-6.

MALONE, T. W.; BERNSTEIN, M. S. **Handbook of Collective Intelligence**. [S.l.]: The MIT Press, 2015. ISBN 0262029812.

MENG, J.; PAULSON, L. C. Lightweight relevance filtering for machine-generated resolution problems. **J. Appl. Log.**, v. 7, n. 1, p. 41–57, 2009. Disponível em: <https://doi.org/10.1016/j.jal.2007.07.004>.

MOURA, L. de; BJØRNER, N. Z3: An efficient smt solver. In: RAMAKRISHNAN, C. R.; REHOF, J. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 337–340. ISBN 978-3-540-78800-3.

NAWAZ, M.; HASAN, O.; NAWAZ, M. S.; VIGER, P. F.; MENG, S. Proof searching in hol4 with genetic algorithm. In: . [S.l.: s.n.], 2020. p. 513–520.

NORELL, U. Towards a practical programming language based on dependent type theory. In: . [S.l.: s.n.], 2007.

PAULIN-MOHRING, C. Introduction to the coq proof-assistant for practical software verification. 01 2012.

PAULSON, L.; BLANCHETTE, J. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: . [S.l.: s.n.], 2015.

PAULSON, L. C.; SUSANTO, K. W. Source-level proof reconstruction for interactive theorem proving. In: SCHNEIDER, K.; BRANDT, J. (Ed.). **Theorem Proving in Higher Order Logics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 232–245. ISBN 978-3-540-74591-4.

PIERCE, B. C. **Types and Programming Languages**. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091.

RIBEIRO, R.; ROGGIA, K.; VASCONCELLOS, C. O uso de assistente de provas no ensino de lógica. In: . [S.l.: s.n.], 2021. p. 05–05.

RICHARDSON, D. Formal systems, logic and semantics. 10 2006.

RUSSELL, B. **Principles of Mathematics**. [S.l.]: Routledge, 1937.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3. ed. [S.l.]: Prentice Hall, 2010.

SABOGAL, G. A. G.; BARCO, A. F.; VILLANUEVA, M. A.; CARDONA, A.; MONTENEGRO, D.; MILLER, S. R.; DIAZ, J. F.; RUEDA, C.; ROY, P. V. The mozart constraint subsystem : system presentation. In: . [S.l.: s.n.], 2013.

SAILLARD, R. **Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice**. Tese (Doutorado), 09 2015.

SARMA, D. N. Distributed artificial intelligence. In: . [S.l.: s.n.], 2011.

SELINGER, P. **Lecture notes on the lambda calculus**. arXiv, 2008. Disponível em: <https://arxiv.org/abs/0804.3434>.

SGHIR, I. **A Multi-Agent based Optimization Method for Combinatorial Optimization Problems**. Tese (Doutorado), 04 2016.

SØRENSEN, M. H.; URZYCZYN, P. **Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)**. USA: Elsevier Science Inc., 2006. ISBN 0444520775.

SULZMANN, M. A general type inference framework for hindley/milner style systems. In: . [S.l.: s.n.], 2001. p. 248–263. ISBN 978-3-540-41739-2.

TONIDANDEL, F.; RILLO, M. Planejamento de ações para automação inteligente da manufatura. **Gest. Prod. vol.9 no.3 São Carlos Dec. 2002**, scielo, v. 9, p. 345 – 361, 12 2002. ISSN 0104-530X. Disponível em: <http://old.scielo.br/scielo.php?script=sci_arttext&pid=S0104-530X2002000300009&nrm=iso>.

TORRENS, P. H. Um cÁlculo de continuaÇÕes com tipos dependentes. 08 2019.

WEISS, G. Multiagent systems: A modern approach to distributed artificial intelligence. In: ____. [S.l.: s.n.], 2000. v. 1, p. –648. ISBN 0262731312.

WEYNS, D.; PARUNAK, V.; MICHEL, F.; HOLVOET, T.; FERBER, J. Environments for multiagent systems. state-of-the-art and research challenges. In: . [S.l.: s.n.], 2004. v. 3374, p. 1–47. ISBN 978-3-540-24575-9.

WOOLDRIDGE, M. **An Introduction to MultiAgent Systems**. 2nd. ed. [S.l.]: Wiley Publishing, 2009. ISBN 0470519460.

YANG, L.-A.; LIU, J.-P.; CHEN, C.-H.; CHEN, Y.-p. Automatically proving mathematical theorems with evolutionary algorithms and proof assistants. 02 2016.

YUSHKOVSKIY, A.; TRIPAKIS, S. Comparison of two theorem provers: Isabelle/hol and coq. 08 2018.

ZULEGER, F. Interactive proof systems. In: . [S.l.: s.n.], 2006.


UFVJM