Provas mecanizadas sobre o algoritmo W já foram apresentadas na literatura, porém nenhuma é a respeito da sua usual implementação, pois dependem de axiomas e não utilizam mônadas para lidar com efeitos. Esta dissertação de mestrado apresenta uma completa implementação e certificação do algoritmo W no assistente de provas Coq, com as provas de consistência e completude da inferência de tipos, assim como terminação, consistência e completude do algoritmo de unificação.

Orientador: Prof. Dr. Cristiano Damiani Vasconcellos

Coorientadora: Prof. Dra. Karina Girardi Roggia

Joinville, 2019

ANO 2019

RAFAEL CASTRO GONÇALVES SILVA | UMA CERTIFICAÇÃO EM COQ DO ALGORITMO W MONÁDICO

UDESC

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

DISSERTAÇÃO DE MESTRADO

UMA CERTIFICAÇÃO EM COQ DO ALGORITMO W MONÁDICO

RAFAEL CASTRO GONÇALVES SILVA

JOINVILLE, 2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM COMPUTAÇÃO APLICADA

RAFAEL CASTRO GONÇALVES SILVA

UMA CERTIFICAÇÃO EM COQ DO ALGORITMO W MONÁDICO

JOINVILLE

2019

RAFAEL CASTRO GONÇALVES SILVA

UMA CERTIFICAÇÃO EM COQ DO ALGORITMO W MONÁDICO

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Computação Aplicada.

Orientador: Dr. Cristiano D. Vasconcellos
Coorientador: Dra. Karina Roggia

JOINVILLE
2019

Ficha catalográfica elaborada pelo programa de geração automática da Biblioteca Setorial do CCT/UDESC, com os dados fornecidos pelo(a) autor(a)

Silva, Rafael Castro Gonçalves
Uma Certificação em Coq do Algoritmo W Monádico / Rafael Castro Gonçalves Silva. -- 2019.
69 p.

Orientador: Cristiano D. Vasconcellos
Coorientadora: Karina Girardi Roggia
Dissertação (mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2019.

1. Inferência de tipos. 2. Algoritmo W. 3. Coq. I. D. Vasconcellos, Cristiano. II. Girardi Roggia, Karina. III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. IV. Titulo.

Uma Certificação em Coq do Algoritmo W Monádico

por

Rafael Castro Gonçalves Silva

Esta dissertação foi julgada adequada para obtenção do título de

Mestre em Computação Aplicada

Área de concentração em "Ciência da Computação",
e aprovada em sua forma final pelo

CURSO DE MESTRADO ACADÊMICO EM COMPUTAÇÃO APLICADA
DO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA
UNIVERSIDADE DO ESTADO DE SANTA CATARINA.

Banca Examinadora:

Dr. Cristiano Damiani Vasconcellos
CCT/UDESC (Orientador/Presidente)

participação por videoconferência

Dr. Arthur Azevedo de Amorim
CMU

Dr. Rodrigo Geraldo Ribeiro
UFOP

Joinville,SC, 22 de agosto de 2019.

AGRADECIMENTOS

Agradeço à minha família e à minha namorada pelo suporte que me deram durante todo o meu mestrado. Agradeço aos meus amigos pelas risadas que amenizaram a monotonia do dia a dia. Agradeço aos meus orientadores por compartilharem as suas experiências comigo. Por fim, agradeço ao órgão de fomento CAPES por ter me fornecido uma bolsa de pesquisa, a qual foi fundamental para a minha dedicação exclusiva no desenvolvimento deste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

“Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set?”

Edsger Dijkstra

RESUMO

Provas mecanizadas sobre o algoritmo W já foram apresentadas na literatura, porém nenhuma é a respeito da sua usual implementação, pois dependem de axiomas e não utilizam mônadas para lidar com efeitos. Esta dissertação de mestrado apresenta uma completa implementação e certificação do algoritmo W no assistente de provas Coq, com as provas de consistência e completude da inferência de tipos, assim como terminação, consistência e completude do algoritmo de unificação.

Palavras-chaves: Inferência de tipos, Algoritmo W, Coq.

ABSTRACT

Mechanized proofs for the type inference algorithm W have already been presented in the literature, however none of them can be said to be similar to a usual implementation since they rely on axioms and don’t use monads to handle effects. In this master thesis we present a complete implementation and certification of algorithm W in Coq, which includes proofs of correctness and completeness of type inference, as well soundness, completeness and termination of the unification algorithm.

Key-words: Type inference, Algorithm W, Coq.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

CCo	Cálculo de Construções
CCI	Cálculo de Construções Indutivas
DM	Damas-Milner

LISTA DE SÍMBOLOS

Γ	Conjunto de suposições/Contexto
τ	Tipo monomórfico
σ	Tipo polimórfico
α	Variável de tipo
β	Variável de tipo
λ	Abstração *lambda*
$\mathbb{S}$	Substituição de tipos
S	Substituição de tipos
FV	Variáveis de tipo livres

SUMÁRIO

1 INTRODUÇÃO

Sistemas de tipos são relações que ditam quais expressões podem ser associadas a quais tipos. Em sistemas formais, como o Cálculo Lambda, os sistemas de tipos são utilizados para evitar paradoxos e computações infinitas. Nas primeiras linguagens de programação, sistemas de tipos serviam apenas para classificar os dados para serem tratados corretamente pelo processador, garantindo o sentido operacional do programa. Com o avanço das linguagens, passaram a ser utilizados na identificação de falhas na consistência de programas. A verificação de tipos expõe não só falhas triviais na lógica do programador, mas também profundos erros conceituais (PIERCE, 2002).

Dentro das linguagens de programação, aquelas que utilizam tipos como uma forma de certificação *a priori* são classificadas como de tipagem estática. Essas linguagens, como C, Java e Haskell, realizam a atribuições de tipo em tempo de compilação. Já as de tipagem dinâmica, como Python, JavaScript e Scheme, somente identificam o tipo de uma variável durante a execução do programa. Geralmente as linguagens de tipagem dinâmica permitem que o tipo de uma variável se altere durante a execução, mas raramente isso é permitido em linguagens de tipagem estática[1].

Muitas vezes essa maior flexibilidade presente nas linguagens de tipagem dinâmica é visto como algo que contribuí para uma maior produtividade, visto que o programador não precisa assinar os tipos das variáveis ou sequer fazer qualquer forma de raciocínio sobre os tipos do seu programa. Porém, essa mesma flexibilidade permite situações com grande potencial de erro, como uma lista em Python com dados de variados tipos: `[1, 'a', objeto, [2]]`. Em linguagens derivadas de Lisp, como Scheme, frequentemente listas com dados heterogêneos são utilizadas para representar estruturas de dados complexas, por exemplo um jeito comum de representar uma árvore binária de busca é por meio de listas: `(4 (1 () ()) (7 (6 () ()) ()))`. A dinamicidade dos tipos em certas linguagens é parte fundamental, de acordo com a intenção dos seus designers, de como as mesmas devem ser usadas para resolver problemas, como mostrou esse último exemplo.

Várias linguagens de tipagem dinâmica contam com *duck typing*, que possibilita dois objetos de tipos distintos serem utilizados como se fossem do mesmo tipo, caso compartilharem os atributos e métodos necessários. Por outro lado, na tipagem estática o polimorfismo possibilita uma forma mais sofisticada de reuso de código. Em especial, o polimorfismo paramétrico utiliza variáveis de tipos as quais podem ser ins-

[1] Um exemplo de exceção é a linguagem Mezzo.

tanciadas para os diferentes usos do objeto/função. Idealmente, a assinatura de tipos da função é a mais geral possível, ou seja, aquela possibilita a maior quantidade de usos distintos da mesma. Por exemplo, em Haskell a função que inverte a ordem dos elementos numa lista tem o tipo mais geral `[a] -> [a]`, pois o tipo dos dados que estão na lista é irrelevante para as permutações. Portanto, esse programa pode ser aplicado a listas com quaisquer tipos de dados, pois a variável de tipo `a` é um parâmetro que pode ser trocado por qualquer outro tipo.

Mesmo nas linguagens de tipagem estática a responsabilidade do programador de assinar os tipos pode ser opcional. Para isso é necessário que o compilador seja capaz de não só verificar os tipos do programa, mas como também fazer a inferência dos mesmos. Na programação funcional, a inferência de tipos é o processo de encontrar a assinatura de tipo de uma expressão, chamada de tipo principal, aquela que melhor representa todos os possíveis usos da mesma[2] . O propósito da inferência de tipos é dar a liberdade ao programador de escolher anotar ou não os tipos das funções. Isso é particularmente interessante quando uma função tem uma assinatura muito complexa ou em situações que é difícil encontrar o tipo mais geral possível devido ao polimorfismo.

O sistema de tipos da linguagem de programação define quais são os programas bem e mal tipados. Os programas mal tipados são, geralmente, aqueles mal formados e sem sentido semântico (cuja execução pode ocasionar um erro), como um programa que soma um número inteiro com um dado do tipo caractere. Quando um sistema de tipos define como bem tipados somente os programas com sentido semântico, diz-se que esse sistema tem a propriedade de *soundness*[3].

Idealmente um sistema de tipos de uma linguagem de programação deve ter a propriedade de *soundness*. Um exemplo clássico de sistema de tipos com a propriedade de *soundness* é o Damas-Miner (DM), que estabeleceu o famoso lema de Robin Milner “Well-typed programs cannot go wrong” (MILNER, 1978). Essa é a mais desejável propriedade de um sistema de tipos de uma linguagem de programação, visto que reduz os possíveis erros que podem acontecer durante a execução de um programa.

Essa propriedade pode ser provada devido a formalização do sistema de tipos e da linguagem de programação como definições matemáticas, as quais não deixam espaço para ambiguidade. Formalizar uma linguagem de programação é, em essência, definir o que ela é por meio das ferramentas da matemática. Isso, quando feito, pode utilizar conhecimentos dos diversos campos da matemática como Gramáticas Formais, Semântica Formal, Teoria dos Conjuntos, Teoria dos Tipos, Teoria dos Domínios, Teoria das Categorias e dentre outras.

[2] Ainda que em certas situações inferir o tipo mais geral não seja possível.
[3] Algumas vezes também chamado de *type safety*.

Um algoritmo de inferência/verificação de tipos idealmente deve respeitar tudo (e nada a mais) o que o seu sistema de tipos define. O algoritmo W é um método de inferência de tipos para o sistema Damas-Milner, cujas provas de consistência (*soundness*) e completude (*completeness*) garantem que o algoritmo estritamente representa as suas regras. O sistema Damas-Milner e o seu algoritmo W são uma das bases de várias linguagens de programação funcional, como Haskell e OCaml.

Alguém poderia perguntar se essas propriedades de sistemas de tipos de linguagens de programação são refletidas diretamente em suas implementações. Infelizmente, isso não acontece visto que o sistema de tipos da maioria das linguagens de programação não é formalizado. A justificativa para isso é razoável: é muito trabalhoso formalizar e provar as propriedades de uma linguagem de programação real, com várias situações complexas e difíceis de provar e, além disso, a validação de provas feitas com papel e caneta é passível de erro humano.

Nos últimos anos, o uso de assistentes de provas trouxe boas expectativas na pesquisa em linguagens de programação, pois não só aproximam a formalização da implementação, mas como também reduzem o trabalho da validação das provas para o apertar de um botão. O *PoplMark Challenge* é um desafio lançado a fim de tornar comum o uso de assisantes de provas em pesquisas sobre linguagens de programação (AYDEMIR et al., 2010). Além disso, em um dos principais simpósios de linguagens de programação, o *Principles of Programming Languages* (POPL), existe uma trilha dedicada ao uso do assistente de provas Coq, chamada CoqPL.

Provas mecanizadas (realizadas em assistentes de provas) de algoritmos de inferência de tipos são bastante complexas e geralmente envolvem milhares de linhas de código. Porém, o trabalho é bem justificado pois é um passo essencial na certificação de compiladores de linguagens de programação funcionais da família do ML. No próximo capítulo apresenta-se uma breve revisão de trabalhos realizados nessa direção.

Esta dissertação de mestrado apresenta uma certificação em Coq de uma completa implementação monádica do algoritmo W, com as provas de consistência e completude. O trabalho apresentado por (DUBOIS; MÉNISSIER-MORAIN, 1999) é a principal fonte de inspiração, pois contém uma formalização do sistema Damas-Milner puro e do algoritmo W. Além da implementação monádica, outras diferenças deste trabalho são incluir uma completa certificação do algoritmo de unificação, utilizar tipos dependentes e automação por $\mathcal{L}$tac.

A técnica que diferencia esta certificação de todas as demais é o uso da *Hoare State Monad* (HSM) (SWIERSTRA, 2009) na implementação, que foi essencial na prova de completude do algoritmo W pois permitiu raciocinar sobre o valor do estado encapsulado na mônada, o qual é utilizado na geração de novas variáveis de tipos (*fresh*). Como mencionado por (KOTHARI; CALDWELL, 2009), as variáveis de tipo *fresh* não são relevantes na prova de consistência, mas são cruciais na completude.

Ainda que o sistema de tipos formalizado neste trabalho, o Damas-Milner, não é o sistema de tipos de nenhuma linguagem de programação moderna de escala industrial[4], acredita-se que as técnicas utilizadas aqui são uma contribuição relevante para esse assunto. Como apontado por (AYDEMIR et al., 2010), para provas longas a necessidade de notação concisa é crucial. Dessa forma, a principal motivação do estudo de caso deste trabalho é defender o uso da HSM como uma forma eficiente de raciocinar sobre algoritmos de inferência de tipos, os quais precisam lidar com efeitos colaterais.

O objetivo geral desta dissertação é a realização de uma certificação do algoritmo W monádico, com os seguintes objetivos específicos:

- Fornecer uma implementação certificada do algoritmo de unificação, com as provas de consistência, completude e terminação, e que seja conveniente de utilizar na certificação do algoritmo W.
- Modificar a *Hoare State Monad* para lidar com o efeito de exceção necessário no algoritmo W e avaliar as qualidades dessa técnica.
- Provar a consistência e a completude do algoritmo W monádico.

Todo o código em Coq deste trabalho está publicamente disponível no link <https://github.com/rafaelcgs10/W-in-Coq>, o qual pode ser facilmente compilado com o comando *make*. Nota-se que o código foi feito para versões 8.9.* do Coq. Dado o alto uso de automação, o Coq demora alguns minutos para verificar todo o código: cerca de 6 minutos num computador *Core i7 3770*. O entendimento de alguns detalhes desta dissertação pode ser facilitado com a leitura acompanhada da execução do código, especialmente onde provas são explicadas.

[4] Por outro lado, várias linguagens desse porte utilizam sistemas de tipos que vieram de Damas-Milner, como Haskell e OCaml.

2 FUNDAMENTOS

Considerando que o objetivo geral deste trabalho é a certificação em Coq do algoritmo W monádico, então este capítulo de fundamentos contém revisões e resumos sobre as questões envolvendo esse objetivo. Explica-se o que é o sistema Damas-Milner e o algoritmo de inferência de tipos W. Lista-se alguns trabalhos relacionados, os quais também apresentam certificações de algoritmos de inferência de tipos. Por fim, faz-se uma breve introdução ao assistente de provas Coq e explica-se a modificação realizada na *Hoare State Monad*.

2.1 SISTEMA DAMAS-MILNER

O Sistema Damas-Milner é um Cálculo Lambda Tipado, com uma forma restrita de polimorfismo por expressões `let`. O sistema foi criado pelo cientista da computação Robin Milner (MILNER, 1978) para a linguagem funcional ML que é uma linguagem de programação de uso geral, mas que inicialmente era apenas uma metalinguagem para o assistente de provas *Logic of Computable Functions* (LCF). Hoje, é a base dos sistemas de tipos de linguagens como OCaml, Miranda e Haskell.

Segundo (DAMAS; MILNER, 1982), a linguagem ML apresentou-se bem-sucedida ao longo dos anos, devido à combinação de flexibilidade (polimorfismo via `let`), robustez (consistência semântica ou *semantic soundness*) e detecção de erros em tempo de compilação.

A Figura 1 contém a gramática informal[1] de DM, onde `e` representa uma expressão, que pode ser composta por variáveis `x`, as quais concretamente podem ser qualquer *string*, aplicações `e e'` e funções anônimas λ`x.e`, sendo `x` o parâmetro e `e` o corpo da função. As variáveis de tipos α também são qualquer *string*, porém geralmente utiliza-se somente as primeiras letras do alfabeto. Tipos (monomórficos) τ são variáveis ou funções (denotado por $\rightarrow$) entre tipos. Os tipos quantificados σ, também chamados de tipos polimórficos ou *schemes*, são tipos que começam com uma sequência finita de variáveis de tipo quantificadas, ou seja, $\forall \alpha . \sigma$ significa que α está quantificado em σ. Por exemplo, o *scheme* $\forall ab.a \rightarrow (b \rightarrow c)$ tem as variáveis a e b quantificadas, mas c está livre (não quantificada).

Uma substituição de tipo, denotada por $\mathbb{S}$, é um mapeamento finito de variáveis de tipo para tipos, representado por uma sequência:

$$[\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, \alpha_3 \mapsto \tau_3, ..., \alpha_n \mapsto \tau_n],$$

[1] Uma gramática informal não segue estritamente a definição de gramáticas formais.

Figura 1 – Gramática informal das expressões e tipos em Damas-Milner.

```
Variáveis           x
Expressões          e ::= x
                      | e e'
                      | λx.e
                      | let x = e in e'

Variáveis de tipo   α
Tipos               τ ::= τ → τ' | α
Schemes             σ ::= ∀α.σ | τ
```

Fonte: o autor. Adaptado de (DAMAS; MILNER, 1982).

onde τ_i são quaisquer tipos e α_i são distintas variáveis de tipo. Também denota-se $\mathbb{S}$ como $[\alpha_i \mapsto \tau_i]$, onde τ_i são os elementos da imagem e α_i do domínio. A substituição identidade, representada por Id, é simplesmente uma sequência vazia. A aplicação de $\mathbb{S}$ num tipo σ é escrita como $\mathbb{S}\sigma$ e definida por:

$$\begin{aligned} \mathbb{S}\alpha_i &\equiv \tau_i, \\ \mathbb{S}\alpha &\equiv \alpha, \text{ se } \alpha \notin \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n\}, \\ \mathbb{S}(\tau \to \tau') &\equiv \mathbb{S}\tau \to \mathbb{S}\tau', \\ \mathbb{S}\forall\alpha.\sigma &\equiv \mathbb{S}'\sigma, \text{ onde } \mathbb{S}' = \mathbb{S} \setminus [\alpha_i \mapsto \_], \end{aligned}$$

onde _ é um símbolo coringa para representar qualquer outro sem ter que nomeá-lo. A composição de substituições é representada por $\mathbb{S}_2 \circ \mathbb{S}_1$, significando aplicar $\mathbb{S}_2$ na imagem de $\mathbb{S}_1$ e adicionar $[\alpha'_i \mapsto \tau'_i]$ de $\mathbb{S}_2$ cujos α'_i não estão no domínio de $\mathbb{S}_1$. Ou seja, considerando $\mathbb{S}_1 = [\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, ..., \alpha_n \mapsto \tau_n]$ e $\mathbb{S}_2 = [\alpha'_1 \mapsto \tau'_1, \alpha'_2 \mapsto \tau'_2, ..., \alpha'_m \mapsto \tau'_m]$, então:

$$\mathbb{S}_2 \circ \mathbb{S}_1 = [\alpha_1 \mapsto \mathbb{S}_2\tau_1, \alpha_2 \mapsto \mathbb{S}_2\tau_2, ..., \alpha_n \mapsto \mathbb{S}_2\tau_n] \cup [\alpha'_i \mapsto \tau'_i]$$

onde $i \in 1..m$ e α'_i não ocorre no domínio de $\mathbb{S}_1$.

O conjunto de todas as variáveis de tipo livres, denotado por FV, é definido recursivamente por:

$$\begin{aligned} FV(\alpha) &= \{\alpha\} \\ FV(\tau \to \tau') &= FV(\tau) \cup FV(\tau') \\ FV(\forall\alpha.\sigma) &= FV(\sigma) \setminus \{\alpha\}. \end{aligned}$$

Uma característica distinguível deste sistema é a presença de *schemes* (tipos quantificados), cujo quantificador $\forall$ sempre ocorre mais a esquerda. Os tipos quantificados são introduzidos numa expressão por meio do `let x = e in e'`[2], onde o

[2] Por isso chama-se polimorfismo via *let*.

tipo de `x` é quantificado para ser utilizado de maneira polimórfica em `e'`. Considere o seguinte programa com a natural extensão para pares:

```
let id = λx.x in (id 1, id 'a'),
```

O tipo quantificado de `id` é instanciado para números e caracteres. A possibilidade de *schemes* adotarem tipos mais específicos gera uma relação de ordem, formalmente definida por:

$$\frac{\tau' \equiv [\alpha_i \mapsto \tau_i]\tau \qquad \beta_i \notin FV(\forall\alpha_1...\alpha_n.\tau)}{\forall\alpha_1...\alpha_n.\tau \geq \forall\beta_1...\beta_m.\tau'}$$

onde β representa variáveis de tipo assim como α.

A leitura dessa regra é feita no sentido horário: num tipo polimórfico $\forall\alpha_1...\alpha_n.\tau$, aplique a substituição $[\alpha_i \mapsto \tau_i]$ em τ e assim obtenha o tipo τ'. Em seguida, as variáveis de tipos α_i quantificadas no tipo original $\forall\alpha_1...\alpha_n.\tau$ são descartadas conforme a substituição $[\alpha_i \mapsto \tau_i]$ e, ao final, resta-se somente as variáveis de tipos quantificadas β_i que não foram eliminadas pela a substituição.

Este sistema de tipos conta com uma especificação formal (ver Figura 2), a qual dita quais tipos podem ser atribuídos a quais expressões. As regras são dadas como julgamentos de tipos da forma $\Gamma \vdash$e $: \sigma$, onde o contexto Γ é um conjunto de suposições composto por duplas de variáveis[3] e tipos, `e` é a expressão sob julgamento e σ é o tipo julgado para `e` sobre Γ. A substituição e a função FV podem ser naturalmente estendidas para serem aplicadas a contextos.

A especificação formal deste sistema de tipos garante que seja realizável a verificação matemática de suas propriedades. Tratando-se de um sistema com polimorfismo, então a propriedade de tipo principal é essencial. Essa propriedade garante que para toda expressão `e` com um julgamento $\Gamma \vdash$ e $: \sigma$, também há um julgamento $\Gamma \vdash$ e $: \sigma'$ tal que $\sigma' \geq \sigma$. Damas e Milner provaram isso indiretamente pelo algoritmo W.

O algoritmo de inferência W (Figura 3) recebe um contexto Γ com uma expressão `e` e retorna o tipo τ de `e` com uma substituição $\mathbb{S}$. Parte do processo consiste em unificar tipos por meio do algoritmo de unificação de (ROBINSON, 1965), usar uma função gen que generaliza (quantifica) os tipos sobre as variáveis que não fazem parte do conjunto de variáveis livres no contexto e, também, instanciar tipos quantificados para expressar o seu polimorfismo. A notação Γ,x $: \tau$ significa incluir o identificador `x` com o tipo τ em Γ. Uma variável de tipo α é dita *fresh* (ou nova) quando ela não foi

[3] Da linguagem de termos e não de tipos.

Figura 2 – Regras de inferência de Damas-Milner (versão *Non-syntax-directed*).

$$\frac{\mathtt{x} : \sigma \in \Gamma}{\Gamma \vdash \mathtt{x} : \sigma}\,(var)$$

$$\frac{\Gamma \vdash \mathtt{e} : \tau \to \tau' \qquad \Gamma \vdash \mathtt{e'} : \tau}{\Gamma \vdash \mathtt{e\ e'} : \tau'}\,(app) \qquad \frac{\Gamma, \mathtt{x} : \tau \vdash \mathtt{e} : \tau'}{\Gamma \vdash \lambda\mathtt{x.e} : \tau \to \tau'}\,(abs)$$

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \Gamma, \mathtt{x} : \sigma \vdash \mathtt{e'} : \tau}{\Gamma \vdash \mathtt{\textbf{let}\ x = e\ \textbf{in}\ e'} : \tau}\,(let)$$

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \mathtt{e} : \forall\alpha.\sigma}\,(gen) \qquad \frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \sigma \geq \sigma'}{\Gamma \vdash \mathtt{e} : \sigma'}\,(spec)$$

Fonte: o autor. Adaptado de (DAMAS; MILNER, 1982).

utilizada pelo o algoritmo ainda. Por fim, nota-se que este algoritmo é consistente e completo com as regras de inferência de DM, estas propriedades são definidas como:

- Consistência (*Soundness*): Se $W(\Gamma, \mathtt{e}) = (\tau, \mathbb{S})$, então é possível provar pelas regras do sistema DM que $\mathbb{S}(\Gamma) \vdash \mathtt{e} : \tau$. Segue-se, também, que é possível derivar $\mathbb{S}(\Gamma) \vdash \mathtt{e} : \mathtt{gen}(\mathbb{S}\Gamma, \tau)$, onde $\mathtt{gen}(\mathbb{S}\Gamma, \tau)$ é o *scheme* computado por W para $\mathtt{e}$ com base em $\mathbb{S}\Gamma$.

- Completude (*Completeness*): Se o sistema DM mostra que e tem o tipo τ, então o algoritmo infere para e o tipo τ', de maneira que $\tau = \mathbb{S}'(\tau')$ para alguma substituição $\mathbb{S}'$. Mais formalmente: dados Γ e $\mathtt{e}$, seja Γ' uma instância de Γ e σ um *scheme* tal que

  $$\Gamma' \vdash e : \sigma,$$

  então

  1. $W(\Gamma, \mathtt{e})$ termina com sucesso.
  2. Se $W(\Gamma, \mathtt{e}) = (\tau, \mathbb{S})$ então, para alguma substituição $\mathbb{S}'$,

  $$\Gamma' = \mathbb{S}'\mathbb{S}\Gamma \text{ e } \mathbb{S}'\mathtt{gen}(\mathbb{S}\Gamma, \tau) \geq \sigma.$$

A propriedade de tipo principal de DM é um corolário da completude do algorítimo W. Da completude de W também é possível derivar que é decidível se uma expressão $\mathtt{e}$ tem algum tipo sobre um contexto Γ, e caso tenha, necessariamente tem um tipo principal sobre Γ. As provas da consistência e da completude do algoritmo W estão presentes na tese de doutorado de Luis Damas (DAMAS, 1984). Essas provas não são um bom ponto de partida para uma certificação mecanizada, pois as regras de inferência utilizadas não são deterministas, o que gera ainda mais casos de prova.

Figura 3 – Algoritmo W: representação sob a forma de função.

$$
\begin{array}{l}
W(\Gamma,\ \texttt{x})\ \texttt{=} \\
\quad \texttt{se}\ \Gamma(\texttt{x})\ \texttt{=}\ \forall\alpha_1...\alpha_n.\tau\ \texttt{então retorna}\ ([\alpha_i \mapsto \tau_i]\tau, Id)\ \texttt{onde}\ \alpha'_i\ \texttt{fresh} \\
\quad \texttt{senão falha} \\
W(\Gamma,\ \texttt{e e'})\ \texttt{=} \\
\quad (\tau, \mathbb{S}_1) \leftarrow W(\Gamma,\ \texttt{e}) \\
\quad (\tau', \mathbb{S}_2) \leftarrow W(\mathbb{S}_1\Gamma,\ \texttt{e'}) \\
\quad \mathbb{S} \leftarrow \texttt{unify}(\mathbb{S}_2\tau, \tau' \rightarrow \alpha)\ \texttt{onde}\ \alpha\ \texttt{fresh} \\
\quad \texttt{retorna}\ (\mathbb{S}\alpha, \mathbb{S} \circ \mathbb{S}_2 \circ \mathbb{S}_1) \\
W(\Gamma,\ \lambda\texttt{x.e})\ \texttt{=} \\
\quad (\tau, \mathbb{S}) \leftarrow W((\Gamma, \texttt{x}\!:\alpha),\ \texttt{e})\ \texttt{onde}\ \alpha\ \texttt{fresh} \\
\quad \texttt{retorna}\ (\mathbb{S}(\alpha \rightarrow \tau), \mathbb{S}) \\
W(\Gamma,\ \texttt{let x = e in e'})\ \texttt{=} \\
\quad (\tau, \mathbb{S}_1) \leftarrow W(\Gamma,\ \texttt{e}) \\
\quad (\tau', \mathbb{S}_2) \leftarrow W(\mathbb{S}_1(\Gamma, \texttt{x}\!:\mathit{gen}(\mathbb{S}_1\Gamma, \tau)), \texttt{e'}) \\
\quad \texttt{retorna}\ (\tau', \mathbb{S}_2 \circ \mathbb{S}_1)
\end{array}
$$

Fonte: o autor. Adaptado de (DAMAS; MILNER, 1982).

Se possível, é sempre preferível reduzir o trabalho da prova minimizando o número de casos.

2.2 TRABALHOS RELACIONADOS

Algumas provas mecanizadas sobre algoritmos de inferência de tipos já foram apresentadas na literatura, abaixo lista-se alguns desses trabalhos:

- (DUBOIS; MÉNISSIER-MORAIN, 1999) forneceram provas em Coq de consistência e completude do algoritmo W. As propriedades da unificação foram tomadas como axiomas e *binding* de variáveis é resolvido com índices de *de Bruijn*.

- (NARASCHEWSKI; NIPKOW, 1999) também apresentaram provas de consistência e completude do algoritmo W, mas em Isabelle/HOL. As propriedades da unificação também foram assumidas como axiomas e *de Bruijn indices* também foi utilizado.

- (URBAN; NIPKOW, 2008) apresentaram provas de consistência e completude do algoritmo W usando Isabelle/HOL com a biblioteca *Nominal Datatype Package*, ao invés de utilizar índices de *de Bruijn* para tratar *binding* de variáveis. A unificação também foi assumida.

- (KOTHARI; CALDWELL, 2009) relataram um resultado parcial na certificação em Coq da extensão polimórfica do algoritmo de Wand[4], o qual foi inicialmente proposto para apenas inferir tipos monomórficos (WAND, 1987) e posteriormente estendido para suportar polimorfismo via *let* (KOTHARI; CALDWELL, 2008). As propriedades do algoritmo de unificação utilizado foram provadas (KOTHARI; CALDWELL, 2009). Nesse relatório parcial, ainda sem polimorfismo, não houve a necessidade de lidar com *binding* de variáveis.

- (TAN; OWENS; KUMAR, 2015) verificaram a inferência de tipos de CakeML, uma linguagem baseada em SML (Standard ML) cujo compilador tem um *backend* completamente verificado (Kiam Tan et al., 2019). O sistema de tipos de CakeML é uma simplificação do presente em SML, onde somente definições no escopo mais externo podem ser polimórficas. As provas de consistência e completude da inferência de tipos foram feitas no assistente de provas HOL4. O algoritmo de unificação utilizado foi verificado (KUMAR; NORRISH, 2010). *Binding* de variáveis também é resolvido com índices de *de Bruijn*. O algoritmo de inferência é baseado no W e implementado por meio de uma mônada de estado-exceção.

- (GARRIGUE, 2015) certificou em Coq o algoritmo de inferência para uma extensão do sistema de tipos de ML, cujas extensões estão presentes em OCaml: polimorfismo estrutural e tipos equirecursivos. A consistência e a principalidade da inferência de tipos foram provadas. As propriedades do algoritmo de unificação também foram provadas. Utilizou-se a técnica proposta por (AYDEMIR et al., 2008) para lidar com *binding* de variáveis, a qual mistura índices de *de Bruijn* com variáveis nominais.

As primeiras certificações de algoritmos de inferência de tipos não contavam com o algoritmo de unificação, visto que isso era um trabalho a parte ainda a se fazer nos respectivos assistentes de provas e que isso poderia ser tomado como axioma. O uso explícito de índices de *de Bruijn* é predominante, apesar de existir algumas bibliotecas que resolvem dificuldades com α-equivalência de variáveis. Esse é um aspecto que precisa ser melhor desenvolvido segundo o *PoplMark Challenge*, uma vez que é uma dificuldade recorrente em provas mecanizadas sobre linguagens de programação (AYDEMIR et al., 2010).

A única certificação monádica é a de CakeML, porém diversos lemas auxiliares sobre a mônada precisaram ser provados, visto que a mônada utilizada é com tipos simples e não garante propriedades do programa relacionadas ao estado. Em comparação ao que foi desenvolvido neste trabalho, esses resultados auxiliares aqui puderam ser evitados. Nos demais trabalhos relacionados, a passagem explícita do

[4] O algoritmo de Wand tem a característica de postergar a unificação de tipos para uma segunda fase, na qual os tipos que precisam ser unificados são dados como um conjunto de restrições.

estado requereu provas relacionando o valor do estado com a chamada da função unificadora.

2.3 UMA INTRODUÇÃO AO COQ

Esta seção é uma breve introdução a assistentes de provas e ao Coq, onde se apresentam as principais vantagens de se utilizar um assistente de provas e em especial o porquê de Coq ser uma boa opção para o desenvolvimento deste trabalho. Os assistentes de provas[5] são programas que auxiliam o desenvolvimento de provas formais, mas diferente de provadores automáticos, não provam teoremas apenas com o apertar de um botão e necessitam que uma pessoa guie a prova. Por esse motivo, assistentes conseguem ter um domínio de uso mais amplo e podem ser utilizados para provar, *a priori*, qualquer resultado que também seria possível com papel e caneta.

Inicialmente provas assistidas/realizadas por programas não foram bem recebidas por matemáticos, pois a função fundamental de uma prova é convencer que algo é verdadeiro e porquê (GEUVERS, 2009). Detalhes da validade de uma prova podem ser ocultados pelo uso destes programas, pois em geral as suas linguagens internas não são de fácil entendimento. Além disso, falhas na teoria e na implementação podem resultar em provadores capazes de provar sentenças falsas. Ao longo dos anos, provas formalizadas em programas conseguiram o reconhecimento da academia, pois empiricamente conseguiram se demostrar ainda mais confiáveis que provas tradicionais. Hoje observa-se que os principais apelos para o uso de assistentes de provas são:

- a verificação mecânica é rápida e evita as falhas humanas;
- interatividade, permite visualizar informações sobre os estados da prova;
- comandos para busca de teoremas e lemas para o progresso da prova;
- automatização de provas com métodos não-deterministas;
- potencialização da capacidade humana de realizar provas;
- extração de programas verificados.

Atualmente existem dezenas de assistentes de provas, alguns deles são: Automath, Agda, Twelf, ACL2, PVS, Minlog, Isabelle, HOL e Coq. Cada assistente é baseado em um formalismo matemático diferente, ou até mesmo em diferentes teorias (Teoria dos Conjuntos, Teoria dos Tipos, Teoria da Prova, entre outras) e possuem

[5] Também chamados de provadores semi-automáticos.

suas peculiaridades que fazem alguns serem mais adequados para certas tarefas do que outros.

O assistente de provas Coq tem como *kernel* (núcleo) o formalismo Cálculo de Construções Indutivas (CCI) (BERTOT; CASTRAN, 2010), uma extensão do Cálculo Lambda Tipado conhecido como Cálculo de Construções (CCo). O CCo é classificado como um Cálculo Lambda polimórfico de ordem superior e com tipos dependentes.

O início do desenvolvimento do assistente Coq ocorreu em 1984 no *Institut national de recherche en informatique et en automatique* (INRIA), com os cientistas Gérard Pierre Huet e Thierry Coquand liderando o projeto. Inicialmente, Coq chamava-se CoC e era uma implementação do mesmo, posteriormente foi estendido para suportar tipos indutivos do CIC e o seu nome foi mudado para Coq. Hoje Coq conta com pesquisadores do mundo todo, que não somente utilizam o assistente, mas também desenvolvem novas ferramentas como a biblioteca *Mathematical Components* (MAHBOUBI; TASSI, 2018), que conta com uma formalização de diversos campos da matemática.

A ideia de utilizar tipos como linguagem de proposições matemáticas e termos lambda como provas vem da conhecida correspondência de Curry-Howard (SØRENSEN; URZYCZYN, 1998), uma relação entre Cálculos Lambda Tipados e sistemas dedutivos da lógica. Coq se sustenta nessa relação para fornecer uma linguagem versátil o suficiente para expressar até mesmo lógicas mais sofisticadas, como as de ordem superior. Parte considerável da matemática pode ser sucintamente expressa em Coq, o que possibilitou a prova de clássicos problemas da matemática, como o Teorema das Quatro Cores (GONTHIER, 2008). Esse teorema, em especial, tem um grande número de casos (633) a serem analisados e seria inviável prová-lo sem o uso das ferramentas de automação de provas que o Coq fornece.

Para a Ciência da Computação, Coq serve principalmente como uma ferramenta de Métodos Formais para verificação de programas, cuja abordagem clássica começa com a escrita do programa. Posteriormente, define-se a sua especificação e, por fim, prova-se que o programa a satisfaz. Outra abordagem possível é especificar os programas com tipos dependentes. Programar com tipos dependentes tem a vantagem de reduzir as três etapas anteriores em uma única, o que ainda pode evitar certas redundâncias se comparada a outra abordagem. A desvantagem da programação com tipos dependentes é principalmente o aumento de complexidade do código, em detrimento da sua legibilidade.

Programas verificados podem ser extraídos para linguagens de programação funcionais, como OCaml e Haskell, que possuem compiladores capazes de gerar códigos executáveis eficientes. Os aspectos não relevantes computacionalmente são descartados no processo de extração, assim uma função com tipos dependentes torna-se uma de tipos simples. A separação do que é computacionalmente relevante e irrelevante é feita, respectivamente, pelos tipos `Set` e `Prop`.

Graças à correspondência de Curry-Howard, Coq é tanto uma linguagem de programação funcional quanto uma linguagem de prova. Mais especificamente, a linguagem do Coq é dividida em quatro partes:

- A linguagem de programação funcional e especificação chamada *Gallina*. Programas e provas escritos em *Gallina* tem a propriedade de *weak normalization*, sempre terminam. *Gallina* basicamente é o CIC.
- A linguagem de comandos *vernacular*, que fornece a interação com o assistente.
- Um conjunto de táticas (*tactics*) para realizar provas, que são traduzidos para termos em *Gallina*.
- A linguagem $\mathcal{L}$tac para implementar novas táticas e realizar automação de provas.

Alguns trabalhos em linguagens de programação que utilizaram Coq já foram comentados nesta dissertação, porém também é relevante mencionar os seguintes: CompCert é uma compilador de C, cujas partes verificadas são livres de “bugs” (LEROY et al., 2012). Simplicity é uma linguagem de programação para *blockchains* com semântica denotacional e operacional definidas em Coq, cuja máquina abstrata pode ser utilizada para derivar limites superiores no uso dos recursos (O’CONNOR, 2017). A Vellvm é uma formalização da LLVM em Coq, que permite a extração de transformações de código verificadas (ZHAO et al., 2012).

Além da *Mathematical Components*, Coq conta com uma variedade de extensões e bibliotecas que podem ser úteis no desenvolvimento de provas complexas. Por exemplo, QuickChick é uma ferramenta de testes aleatórios baseados em propriedades, que serve para encontrar contra-exemplos em proposições que parecem verdadeiras, mas na realidade não são (PARASKEVOPOULOU et al., 2015). A biblioteca Libtactics conta com um revisão do conjunto de táticas padrão do Coq, que são mais convenientes para provar propriedades de linguagens de programação.

Outra vantagem de utilizar Coq são os materiais didáticos disponíveis gratuitamente na internet: *Software Foundations* (PIERCE et al., 2017) e *Certified Programming with Dependent Types* (CHLIPALA, 2013). O primeiro conta, atualmente, com quatro volumes focando respectivamente em fundamentos da lógica, teoria de linguagens de programação, verificação de algoritmos funcionais e testes baseados em propriedades com QuickChick. O segundo livro tem como objetivo o desenvolvimento de software verificado, usando tipos dependentes e intenso uso de automação de provas. Chlipala advoca por um estilo de provas onde a maior parte do trabalho é automatizado por scripts em $\mathcal{L}$tac.

Portanto, Coq é fruto de anos de pesquisa em sistemas de tipos e assistentes de provas, e vem se demonstrando adequado para pesquisas sobre linguagens de programação, pois conta com bons materiais didáticos, extensões e bibliotecas para esse fim.

2.4 PROVAS SOBRE PROGRAMAS MONÁDICOS

A escrita de programas com efeitos colaterais numa linguagem de programação funcional pura pode levar a definições com vários argumentos e, por consequência, torna a programação funcional mais complexa. Linguagens como Haskell resolveram esse problema por meio da passagem implícita de dados através das mônadas. As mônadas foram introduzidas na programação funcional por (WADLER, 1992), como um conceito importado da Teoria das Categorias.

Provar em Coq uma propriedade de um programa com efeito é factível, porém o excesso de argumentos não só torna a escrita do programa menos sucinta, mas também aumenta a complexidade de sua especificação e lemas. Como exemplo, considere a definição de árvore binária de números naturais e a definição de novo número (na árvore), presentes na Figura 4. Deseja-se, então, um programa que substitui os números zeros na árvore por novos números ainda não presentes na mesma.

A solução apresentada pela função `change_zeros`, com tipos dependentes, exige uma prova de `NewNumberTree t n` como argumento, onde `t` é a árvore e `n` é um número que não ocorre em `t`. O retorno da função é um elemento do tipo `sig`, que consiste em um par `(t', n')` acompanhado de uma prova da seguinte propriedade desejada: `n'` é novo em `t'` e `n` é menor ou igual a `n'`. A condição `n <= n'` diz que o valor do argumento `n` somente pode crescer ou se manter igual e auxilia na prova de `NewNumberTree t' n'` no caso recursivo. As obrigações de provas geradas por `Program` são simples de resolver, porém não somente neste exemplo mas em todo o restante desta dissertação as provas em Coq foram omitidas a fim de reduzir o uso de espaço pelas figuras.

Essa solução é correta, porém uma definição monádica seria mais natural. Evidentemente, escrever programas monádicos em Coq é uma possibilidade, basta definir o tipo da mônada, e as suas respectivas funções `return` e `bind`. Porém, o dado passado implicitamente pela mônada não é apresentado no contexto da prova, o que torna a prova da propriedade do exemplo acima impossível numa definição monádica. A informação escondida pela mônada é justamente a mais fundamental na prova.

Figura 4 – Exemplo de prova sobre um programa com efeito de estado.

```
Inductive TreeNat : Type :=
| Leaf : nat -> TreeNat
| Node : TreeNat -> TreeNat -> TreeNat.

Inductive NewNumberTree : TreeNat -> nat -> Prop :=
| LeafNew : forall n1 n2, n1 < n2 ->
                    NewNumberTree (Leaf n1) n2
| NodeNew : forall t1 t2 n, NewNumberTree t1 n ->
                           NewNumberTree t2 n ->
                           NewNumberTree (Node t1 t2) n.

Program Fixpoint change_zeros (t : TreeNat) (n : nat) :
 NewNumberTree t n -> {(t', n') : TreeNat * nat | NewNumberTree t' n' /\ n <= n'} :=
 fun pre =>
 match t with
  | Leaf 0 =>  exist _ (Leaf n, S n) _
  | Leaf n' => exist _ (Leaf n', n) _
  | Node t1 t2 => match change_zeros t1 n _ with
                | (t1', n') => match change_zeros t2 n' _ with
                              | (t2', n'') => exist _ (Node t1' t2', n'') _
                              end
                end
 end.
```

Fonte: o autor.

2.4.1 A Mônada de Estado de Hoare

Uma solução dessa dificuldade é a Mônada de Estado de Hoare, ou *Hoare State Monad* (HSM), apresentada por (SWIERSTRA, 2009) como uma variante da usual mônada de estados, a qual pode ser facilmente aplicada a verificação de programas com efeitos de estado em Coq. Essa mônada, em Coq, é uma definição de um tipo `HoareState p a q`, tal que os parâmetros são em ordem: a pré-condição, o tipo do valor de retorno e a pós-condição (ver Figura 5). Essa formulação permite raciocinar sobre programas na mônada de estado conforme o estilo de Hoare-Floyd.

O dado do estado é parametrizado como `st` em `Set`. A definição `Pre` garante que o estado inicial satisfaz alguma pré-condição. A definição `Post` assegura que o par de retorno satisfaz alguma pós-condição relacionada ao estado inicial, o valor de retorno e o estado final. O tipo da mônada `HoareState` aceita todos os estados iniciais `i` que satisfazem uma dada pré-condição e retorna um par `(x, f)` que satisfaz uma dada pós-condição. Para os casos onde a pré-condição não tem restrição, então usa-se a definição `top`.

Para a função `ret` (`return`) a pré-condição é apenas `top`, mas a sua pós-condição deve garantir que o valor colocado dentro da mônada é o argumento `x` e que o estado não se alterou. O operador `bind` é sobre compor computações monádicas

Figura 5 – A Mônada de Estado de Hoare.

```
Variable st : Set.

Definition Pre : Type := st -> Prop.

Definition Post (a : Type) : Type := st -> a -> st -> Prop.

Program Definition HoareState (pre : Pre) (a : Type) (post : Post a) : Type :=
  forall i : {t : st | pre t}, {(x, f) : a * st | post i x f}.

Definition top : Pre := fun st => True.

Program Definition ret (a : Type) : forall x,
    @HoareState top a (fun i y f => i = f /\ y = x) := fun x s => (x, s).

Program Definition bind : forall a b P1 P2 Q1 Q2,
    (@HoareState P1 a Q1) -> (forall (x : a), @HoareState (P2 x) b (Q2 x)) ->
    @HoareState (fun s1 => P1 s1 /\ forall x s2, Q1 s1 x s2 -> P2 x s2)
                b
                (fun s1 y s3 => exists x, exists s2, Q1 s1 x s2 /\ Q2 x s2 y s3) :=
                 fun a b P1 P2 Q1 Q2 c1 c2 s1 => match c1 s1 as y with
                                  | (x, s2) => c2 x s2
                                  end.
```

Fonte: o autor. Adaptado de (SWIERSTRA, 2009).

e, aqui, o seu tipo é um tanto mais complexo que o usual `M a -> (a -> M b) -> M b` para alguma mônada `M`. A primeira ideia chave para o seu tipo é como a pré-condição (`P2`) e a pós-condição (`Q2`) da segunda computação (`c2`) podem referenciar o resultado da primeira computação (`c1`). A solução é fazer ambas condições dependerem de um valor `x` do tipo `a` presente na primeira computação:

```
forall (x : a), @HoareState (P2 x) b (Q2 x).
```

A segunda ideia chave para esse tipo são a pré-condição e a pós-condição do retorno, isso é, as condições que a composição das duas computações deve satisfazer. Certamente, essa pré-condição deve satisfazer a pré-condição de `c1` (`P1` para algum estado inicial `s1` e também é necessário que a pós-condição de `c1` (`Q1`) implique na pré-condição de `Q2`. Logo, a pré-condição da computação composta é:

```
fun s1 => P1 s1 /\ forall x s2, Q1 s1 x s2 -> P2 x s2.
```

A pós-condição da computação composta deve garantir ambas as pós-condições `Q1` e `Q2`. No entanto, isto não pode diretamente referenciar o valor e o estado intermediários. Então, para resolver isso, quantifica-se existencialmente sobre os resultados da primeira computação:

```
fun s1 y s3 => exists x, exists s2, Q1 s1 x s2 /\
                                    Q2 x s2 y s3.
```

Ainda que o tipo do operador `bind` seja um tanto complexo, a sua definição é apenas a usual para a mônada de estados. Após completamente definir `bind` é necessário resolver algumas obrigações de provas sobre `s2` (o estado final da primeira computação e inicial da segunda) satisfazer a pré-condição da segunda computação e a aplicação `c2 x s2` satisfazer a pós-condição da computação composta.

Funções como `put` e `get` são triviais de definir, então elas são omitidas aqui. Em Coq é possível definir a notação *do*, que permite utilizar o combinador `bind` para escrever programas num estilo similar ao do paradigma imperativo:

```
Infix ">>=" := bind (right associativity, at level 71).

Notation "x <- m ; p" := (m >>= fun x => p)
    (at level 68, right associativity,
     format "'[' x  <-  '[' m ']' ; '//' '[' p ']' ']'").
```

Assim, voltando ao exemplo proposto na Figura 4, a definição de `change_zeros` pode ser feita de maneira monádica conforme mostra a Figura 6. A função `fresh` computa um novo número ao retornar o estado atual e incrementa-lo. A pré-condição de `change_remove_m` espera que o estado inicial (atual) da mônada seja um valor superior a todos na árvore `t`. Já a pós-condição exige que o estado final seja um número novo na árvore a ser retornada. A pós-condição `i <= f` facilita a prova de `NewNumberTree t' f`.

Figura 6 – Exemplo de prova sobre um programa com efeito de estado.

```
Program Definition fresh : @HoareState nat
  (@top nat) nat (fun i x f => S i = f /\ i = x) := fun n => (n, S n).

Program Fixpoint remove_zeros_m (t : TreeNat) :
  @HoareState nat (fun i => NewNumberTree t i) TreeNat
   (fun i t' f => NewNumberTree t' f /\ i <= f) :=
  match t with
  | Leaf 0 =>
          n' <- fresh ;
          ret (Leaf n')
  | Leaf n' => ret (Leaf n')
  | Node t1 t2 =>
              t1' <- remove_zeros_m t1 ;
              t2' <- remove_zeros_m t2 ;
              ret (Node t1' t2')
  end.
```

Fonte: o autor.

2.4.2 A Mônada de Exceção-Estado de Hoare

Conforme apontado por (GIBBONS; HINZE, 2011) não há como aplicar diretamente a HSM aos demais efeitos, logo faz-se necessário adaptá-la para o uso deste trabalho. O algoritmo W tem dois tipos de efeitos: estado e exceção. A HSM somente lida com estado, portanto modifica-se a HSM para também capturar erros: embute-se o tipo `sum`[6] na definição de `HoareState` (ver Figura 7). A nova mônada chama-se Mônada de Exceção-Estado de Hoare, ou *Hoare Exception-State Monad* (HESM). Essa modificação não cria muita complexidade adicional, logo todos os detalhes explicados na subseção anterior ainda são válidos.

O tipo da `HoareState` modificada tem um parâmetro adicional `B:Prop` para o lado direito do tipo `sum`. O esse novo tipo diz que para qualquer estado inicial que satisfaz uma dada pré-condição, existe um dado do tipo `sum` cujo lado esquerdo satisfaz uma dada pós-condição e cujo lado direito não tem condições. As obrigações de provas geradas por `Program` na nova definição de `bind` são tão fáceis quanto as anteriores. O parâmetro `B` servirá na implementação do algoritmo W para armazenar o tipo que representa todos os possíveis erros/exceções do algoritmo.

Figura 7 – A Mônada de Exceção-Estado de Hoare.

```
Program Definition HoareState (B:Prop) (pre:Pre) (a:Type) (post:Post a) : Type :=
  forall i : {t : st | pre t},
    {e : sum (prod a st) B | match e with
                             | inl (x, f) => post (proj1_sig i) x f
                             | inr _ => True
                             end}.

Program Definition ret {B:Prop} (a:Type) : forall x,
  @HoareState B top a (fun i y f => i = f /\ y = x) :=
    fun x s => exist _ (inl (x, s)) _.

Program Definition bind : forall a b P1 P2 Q1 Q2 B,
  (@HoareState B P1 a Q1) -> (forall (x:a), @HoareState B (P2 x) b (Q2 x)) ->
  @HoareState B (fun s1 => P1 s1 /\ forall x s2, Q1 s1 x s2 -> P2 x s2) b
              (fun s1 y s3 => exists x, exists s2, Q1 s1 x s2 /\ Q2 x s2 y s3) :=
    fun B a b P1 P2 Q1 Q2 c1 c2 s1 => match c1 s1 as y with
                                      | inl (x, s2) => c2 x s2
                                      | inr R => _
                                      end.
```

Fonte: o autor.

2.5 AUTOMAÇÃO EM COQ

Durante a prova de propriedades de linguagens de programação é comum que muitos casos de prova sejam apenas repetições tediosas e com poucas diferen-

[6] O tipo `Either` em outras linguagens de programação funcional.

ças entre elas. Naturalmente, espera-se que tais repetições possam ser resolvidas de maneira automática, visto que uma das funções de um assistente de provas é a semi-automatização. Além de táticas como `auto`, `congruence` e `omega`, as quais automatizam a solução de certos tipos de provas, Coq conta com a linguagem de script $\mathcal{L}$tac para o desenvolvimento de novas táticas com a finalidade de automação.

A prova mecanizada da unificação fornecida por (RIBEIRO; CAMARÃO, 2016) utiliza uma própria tática personalizada chamada `mysimp`, que foi construída com $\mathcal{L}$tac e com algumas táticas da biblioteca *LibTactic* desenvolvida por Arthur Charguraud (PIERCE et al., 2017). Essa biblioteca fornece um conjunto de táticas específicas para o desenvolvimento de provas sobre linguagens de programação e, portanto, também é utilizada neste trabalho. A tática personalizada `mysimp` é um procedimento que tenta simplificar o estado da prova ao aplicar diversas manipulações nas hipóteses por meio de uma outra $\mathcal{L}$tac chamada `s`, então, tenta resolver o objetivo com `auto` (ver Figura 8).

Figura 8 – A tática personalizada `mysimp`.

```
Ltac s :=
  match goal with
           ⋮
    | [ H : _ \/ _ |- _] => destruct H
    | [ |- context[eq_id_dec ?a ?b] ] => destruct (eq_id_dec a b);
                                           subst ; try congruence
    | [ H : (_,_) = (_,_) |- _] => inverts* H
    | [ H : Some _ = Some _ |- _] => inverts* H
           ⋮
  end.

Ltac mysimp := (repeat (simpl; s)) ; simpl; auto with arith.
```

Fonte: o autor.

A tática personalizada aplicada nas provas deste trabalho é uma extensão de `mysimp`, chamada `crush`, para tentar não somente simplificações do estado, mas para também tentar provas por reescrita e a resolução por `eauto` e `omega` (ver Figura 9). Para a simplificação do estado da prova utiliza-se a $\mathcal{L}$tac `crush'`, que é similar a `s`, mas faz *match* em construtores a mais.

A $\mathcal{L}$tac `rewriter` para reescrita é a mesma utilizada no livro (CHLIPALA, 2013), a qual utiliza a tática `autorewrite` para tentar resolver o objetivo reescrevendo onde for possível (visto o `in *`) com os lemas equacionais do `Hint` *data-base* `RE`[7]. Além disso, a $\mathcal{L}$tac `rewriteHyp` procura hipóteses para reescrever o objetivo, desde que todos os novos sub-objetivos sejam resolvidos.

7 Lemas são adicionados a essa *data-base* com o *vernecular* `Hint Rewrite nome_do_lema : RE`.

Figura 9 – A tática personalizada `crush`.

```
Ltac rewriteHyp :=
  match goal with
    | [ H : _ |- _ ] => rewrite H by solve [ auto ]
  end.

Ltac rewriterP := repeat (rewriteHyp; autorewrite with RE in *).
Ltac rewriter := autorewrite with RE in *; rewriterP; eauto; fail.

Ltac crush := repeat (intros; simpl in *; try crush'; subst); eauto;
              try contradiction; try splits; try omega; try rewriter;
              autorewrite with RELOOP using congruence.
```

Fonte: o autor.

Para evitar um sistema de reescrita que possa entrar em loop, utilizou-se um segundo `Hint` *data-base* `RELOOP` com a tática `autorewrite`, mas com `congruence` após cada reescrita, o que ajuda a evitar *loops*.

Apesar de ser ineficiente em diversas situações devido ao custo da reescrita tanto no objetivo quanto nas hipóteses, a tática personalizada `crush` resolveu muitos casos de prova neste trabalho que a tática `mysimp` não conseguiu.

3 CERTIFICAÇÃO DO ALGORITMO DE UNIFICAÇÃO

O algoritmo para a unificação de fórmulas de primeira ordem foi apresentado por (ROBINSON, 1965) e vem sendo utilizado por algoritmos de inferência de tipos desde os trabalhos de (HINDLEY, 1969) e (MILNER, 1978). A unificação pode ser utilizada na inferência de tipos de duas formas. A primeira, relacionada com as ideias dos últimos dois autores, corresponde aos algoritmos de uma única fase, os quais usam a unificação (geralmente no caso da aplicação) para encontrar uma substituição S para um par de tipos τ, τ', tal que a aplicação de S em ambos os tipos resulta em $S\tau, S\tau'$ sintaticamente iguais.

A segunda abordagem advém das ideias de (WAND, 1987) (o algoritmo de Wand), (REMY et al., 1992) e (SULZMANN; ODERSKY; WEHR, 1999). Esses são os algoritmos de duas fases: na primeira fase não há unificação de tipos, então o algoritmo em questão retorna uma assinatura de tipo τ'' e um conjunto de equações E entre tipos. A segunda fase resolve essas equações para encontrar uma substituição S, tal que para toda equação $\tau = \tau' \in E$ a aplicação de S em ambos os lados resulta na igualdade $S\tau = S\tau'$ sendo verdadeira. Finalmente, a substituição S é aplicada em τ'' resultando no tipo inferido. O primeiro estilo de inferência é chamado de *substitution-based* e o segundo de *constraint-based* (KOTHARI; CALDWELL, 2008).

Provas mecanizadas em Coq do algoritmo de unificação foram desenvolvidas por (RIBEIRO; CAMARÃO, 2016), no qual difere das demais certificações[1] por seguir o estilo de prova presente em livros-texto sobre linguagens de programação, como (MITCHELL, 1996; PIERCE, 2002). A formulação do algoritmo nesses livros-texto é a tradicional/comum, já as provas citadas no rodapé 1 fazem modificações no algoritmo, como torná-lo de recursão estrutural. A unificação utilizada neste trabalho é apenas uma modificação do código fornecido por Ribeiro e Camarão, o qual refere-se aqui como a *unificação original*. Muitas das definições aqui são exatamente as mesmas da unificação original, por exemplo, a definição de tipo (ver Figura 10). Os identificadores de variáveis são apenas números naturais, por uma questão de simplicidade e, também, para o uso de índices de *de Bruijn* quando for necessário lidar com tipos quantificados.

A modificação feita na unificação original é a fim de tornar o algoritmo mais próximo da sua usual implementação para o algoritmo W. Portanto, ao invés de unificar uma lista de equações de tipo, a versão modificada apenas unifica um par de tipos. Ainda que essa modificação pareça ser cosmética, uma vez que uma formula-

[1] Por exemplo, (PAULSON, 1984; BOVE, 1999; MCBRIDE, 2003; KOTHARI; CALDWELL, 2010)

ção pode facilmente simular a outra, isso tem impacto em como formular a prova de terminação. Além disso, a aplicação da substituição de tipos é definida de uma forma mais conveniente para a prova de completude do algoritmo W.

3.1 SUBSTITUIÇÃO DE TIPOS

Como já definido na Seção 2.1, a substituição de tipos é um mapeamento finito de variáveis de tipo para tipos. A literatura sobre implementação de sistemas de tipos para linguagens de programação apresenta diversas formas de como definir a substituição de tipos e as suas operações (lista de associatividade, dicionário, tipo funcional, etc). Quando definido como uma lista de associatividade, a operação de aplicação algumas vezes é definida incrementalmente, isso é:

$$S(\tau) = \begin{cases} \tau & \text{se } S = [\,] \\ S'([\alpha \mapsto \tau']\,\tau) & \text{se } S = [\alpha \mapsto \tau'] :: S' \end{cases}$$

Essa definição incremental foi utilizada na unificação original, pois a mesma foi baseada nos livros-texto de Mitchell e Pierce (MITCHELL, 1996; PIERCE, 2002). Por outro lado, outros autores definem a aplicação de uma forma não-incremental[2], que é o estilo escolhido para este trabalho (ver Figura 10), onde `find_subst` é apenas uma função *look-up*. A definição não-incremental é mais próxima da apresentada na Seção 2.1 e das referências clássicas sobre algoritmos de inferência (DAMAS, 1985).

Ambas as definições de aplicação, incremental e não-incremental, podem representar todos os possíveis mapeamentos entre variáveis de tipo e tipos, embora as duas sejam extensionalmente diferentes na aplicação. Por exemplo, considerando uma substituição $[a \mapsto b, b \mapsto d]$ e um tipo $a \to b$, na aplicação não-incremental o resultado é $b \to d$ e na incremental é $d \to d$. O objetivo dessa alteração é conseguir alguns lemas para a prova de completude do algoritmo W, os quais somente existem na definição não-incremental, por exemplo:

```
forall (i:id) (t:ty) (s:substitution),
  apply_subst ((i, t)::s) (var i) = t.
```

A composição de substituições segue o mesmo conceito apresentado na Seção 2.1: aplica-se a segunda substituição (`s2`) na imagem da primeira (`s1`) e adiciona-se os mapeamentos de variáveis de `s2` que não estão em `s1`. No entanto, `find_subst` está preocupado apenas com a primeira ocorrência de dado um identificador, então não é necessário evitar variáveis repetidas no domínio, por consequência a definição de composição é:

[2] Por exemplo, (JONES, 1987; JONES, 1999; GRABMÜLLER, 2006; DIEHL, 2015).

Figura 10 – Definição de tipos, substituição e aplicação de substituição.

```
Definition id := nat.

Inductive ty : Set :=
| var : id -> ty
| con : id -> ty
| arrow : ty -> ty -> ty.

Definition substitution := list (id * ty).

Fixpoint apply_subst (s:substitution) (t:ty) : ty :=
  match t with
  | arrow l r => arrow (apply_subst s l)
                      (apply_subst s r)
  | var i => match find_subst s i with
             | None => var i
             | Some t' => t'
             end
  | con i => con i
  end.
```

Fonte: o autor.

```
Definition compose_subst (s1 s2 : substitution) :=
  apply_subst_list s1 s2 ++ s2.
```

onde `apply_subst_list` aplica a primeira substituição na imagem da segunda. Na unificação original, a composição de substituições é apenas a concatenação de listas (`++`) e, assim, é trivial a igualdade:

```
Lemma apply_compose_equiv : forall s1 s2 t,
  apply_subst (compose_subst s1 s2) t = apply_subst s2 (apply_subst s1 t).
```

a qual também é verdadeira na aplicação não-incremental.

3.2 O ALGORITMO DE UNIFICAÇÃO

O algoritmo de unificação segue a sua tradicional implementação numa linguagem de programação funcional, parcialmente mostrada pela Figura 11. Detalhes sobre `constraints` e `constraints_lt` estão na Seção 3.3 sobre a terminação, e detalhes sobre o tipo (especificação) `unify_type` na Seção 3.4 sobre a certificação. Outras definições ainda não explicadas também o serão nas próximas seções.

Poder-se-ia esperar que este algoritmo fosse diretamente implementado na HESM, com a especificação `unify_type` na pós-condição da mesma, porém isso iria trazer complicações para a prova de terminação: a obrigação de prova sobre o argumento de terminação (`constraints_lt`) na segunda chamada recursiva não teria acesso a dados importantes, relativos à `unify_type`, da primeira chamada recursiva.

Figura 11 – O algoritmo de unificação.

```
Program Fixpoint unify' (l : constraints) {wf constraints_lt l} : unify_type l :=
fun wfl =>
  match get_tys l with
                                    ⋮
  | (var i, t) => match occurs_dec i t with
                  | left _ => inr _ _
                  | right _ => if (eq_ty_dec (var i) t)
                              then inl _ (@exist substitution _ nil _)
                              else inl _ (@exist substitution _ ((i, t)::nil) _)
                  end
  | (arrow l1 r1, arrow l2 r2) =>
                match unify' (mk_constraints (get_ctxt l) l1 l2) _ with
                | inr _ E => inr _ _
                | inl _ (exist _ s1 HS) =>
                     match unify' (mk_constraints
                                   (minus (get_ctxt l) (dom s1))
                                   (apply_subst s1 r1) (apply_subst s1 r2)) _ with
                     | inr _ E => inr _ _
                     | inl _ _ (exist _ s2 HS') =>
                               inl _ (exist substitution _ (compose_subst s1 s2) _)
                     end
                end
  end.
```

Fonte: o autor.

A opção `wf constraints_lt l` indica que a terminação será provada pela relação bem fundada `constraints_lt` sobre o termo `l`.

3.2.1 *Occurs Check*

O algoritmo de unificação utiliza o bem conhecido *occurs check* para evitar a geração de mapeamentos cíclicos, como $[\alpha_i \mapsto \alpha_i]$. Assim, define-se o predicado `occurs v t` que é verdadeiro somente se o identificador `v` ocorre em alguma variável do tipo `t`. A verificação `occurs_dec` é um procedimento de decisão com tipos dependentes, o qual fornece uma prova de que ou o identificador `i` ocorre no tipo `t` ou não: `{occurs v t} + {~ occurs v t}`.

3.3 TERMINAÇÃO DO ALGORITMO DE UNIFICAÇÃO

Uma importante e necessária propriedade de Gallina é a garantia da terminação de todas as suas funções definíveis. O critério padrão para terminação de Gallina é um tanto conservador, visto que pode ser realizado com uma simples verificação sintática. Esse critério é conhecido como recursão estrutural. As funções de recursão primitiva, um subconjunto de todas as funções totais, executam chamadas recursivas em sub-termos sintáticos do argumento utilizado no critério de parada.

O algoritmo de unificação faz uma chamada recursiva em termos que não são sub-estrutura do (possível) argumento para terminação: `(apply_subst s1 r1)` e `(apply_subst s1 r2))`. Ainda, é possível que os tipos aumentem de tamanho com a substituição de tipos e como consequência não é óbvia a convergência da recursão. A prova de terminação clássica para este algoritmo, usada na certificação de Ribeiro e Camarão, é baseada numa relação bem-fundada sobre uma ordem lexicográfica. As mudanças feitas neste trabalho são poucas e, em essência, a prova de terminação é a mesma.

A primeira parte da ideia para a prova de terminação é o uso de restrições de tipo (*type constraints*). Na unificação original, isso é definido como um produto dependente de um contexto de variáveis $\mathbb{V}$ (uma lista de variáveis de tipo) e uma lista de pares de tipos (restrições a serem resolvidas)[3]. O contexto de variáveis contém, a cada passo da recursão, o complemento do conjunto de variáveis de tipos que estão no domínio da substituição unificadora (*unifier*) naquele passo. Considerando que neste trabalho não se unifica uma lista de pares de tipos, mas sim apenas um par de tipos, então as restrições de tipo são simplificadas, conforme a Figura 12, para apenas um par de tipos.

Figura 12 – Definição de restrições de tipo e as suas funções auxiliares.

```
Definition constraints := sigT
                (fun _ : varctxt => (ty * ty)%type).
Definition get_ctxt (C:constraints) : varctxt
                            := let (v,_) := c in v.
Definition get_tys (C:constraints) : (ty * ty)%type
                            := let (_,l) := c in l.
Definition mk_constraints (V:varctxt) (t1 t2:ty)
                : constraints := existT _ V (t1, t2).
```

Fonte: o autor.

[3] Na unificação original, duas coisas distintas são chamadas de restrições: o produto dependente `constraints` e a lista de pares de tipos a serem unificados, no qual o segundo é parte do primeiro. Este trabalho herdou (na implementação) chamar o par de tipos a ser unificado de restrições, porém evita-se fazê-lo por clareza.

Como mencionado acima, o contexto de variáveis $\mathbb{V}$ obedece relações com os demais elementos da unificação: o par de tipos τ, τ' a ser unificado e, também, a substituição unificadora S. Ao todo são três relações a serem definidas, chamadas de condições de boa-formação (*well-formed conditions*), tais que:

- Um tipo τ é bem-formado em relação a $\mathbb{V}$ se todas as variáveis de tipo de τ estão em $\mathbb{V}$.
- Uma substituição $S = \{[\alpha \mapsto \tau] :: S'\}$ é bem-formada em relação a $\mathbb{V}$ se:
  1. α está em $\mathbb{V}$
  2. τ é bem-formado em respeito a $\mathbb{V} - \{\alpha\}$
  3. τ é bem-formado em respeito a $\mathbb{V} - dom(S')$
  4. S' é bem formado em respeito a $\mathbb{V} - \{\alpha\}$
- Um par de tipos é bem-formado se ambos os tipos são bem-formados.

Essas três definições são fáceis de expressar em Coq como `Fixpoint` retornando `Prop` e são nomeadas, respectivamente, `wf_ty`, `wf_subst` e `wf_constraints`[4]. A terceira condição em `wf_subst` é uma adição em relação à unificação original, a qual foi necessária devido ao seguinte lema essencial para a prova de terminação:

```
Lemma substs_remove :
  forall (t:ty) (s:substitution) (V:varctxt),
         wf_subst V s -> wf_ty V t ->
         wf_ty (minus V (dom s)) (apply_subst s t).
```

que é verdadeiro para a substituição de tipo incremental sem a condição adicional, mas se torna falso na operação não-incremental. Essa condição adicional não compromete qualquer aspecto do funcionamento do algoritmo. O *Quickchick* foi utilizado para verificar a quebra desse lema e também para descobrir a condição adicional para o recuperar. A infraestrutura básica do *Quickchick* foi suficiente para os testes, apenas foi necessário escrever o gerador para o tipo indutivo `ty`, que é uma estrutura em árvore bastante simples.

A ordem lexicográfica combina o tamanho dos tipos a serem unificados com o número de elementos no contexto de variáveis $\mathbb{V}$. A recursão segue a ordem lexicográfica, tal que ou o par de tipos reduz de tamanho ou número de elementos em $\mathbb{V}$ reduz. Em ambos os casos a recursão sempre avança para um elemento menor nessa ordenação. As definições de tamanho de tipos, de pares de tipo, e da ordem

[4] O par de tipos a ser unificado também é chamado de restrições.

lexicográfica estão na Figura 13. A função `lexprod` permite a definição de ordem lexicográfica sobre pares dependentes, neste caso o tipo `constraints`, no qual o primeiro elemento da ordenação é o tamanho de $\mathbb{V}$ e o segundo é o tamanho do par de tipos.

Figura 13 – Definições de tamanho de tipo, de par de tipos e de ordem lexicográfica.

```
Fixpoint size (t:ty) : nat :=
  match t with
    | arrow l r => 1 + size l + size r
    | _ => 1
  end.

Definition size_t (t:(ty * ty)) : nat :=
  match t with
  | (t1, t2) => size t1 + size t2
  end.

Definition constraints_lt : constraints -> constraints -> Prop :=
  lexprod varctxt (fun _ => (ty * ty)%type)
          (fun (x y : varctxt) => length x < length y)
          (fun (x : varctxt) (t t' : (ty * ty)%type) => size_t t < size_t t').
```

Fonte: o autor.

A demostração que a relação de ordem `constraints_lt` é bem-fundada é derivada facilmente, pois a biblioteca de ordem lexicográfica fornece um combinador `wf_lexprod`. Esse combinador junta duas relações bem-fundadas, uma simples e outra de pares dependentes, para formar uma relação bem-fundada de pares por uma ordem lexicográfica.

O resto da prova de terminação é demostrar que a recursão obedece a ordem lexicográfica e preserva a propriedade bem-formada. Para a primeira chamada recursiva é trivial de provar, visto que é uma chamada em termos de tamanhos menores e o contexto de variáveis V é o inicial. A segunda chamada recursiva é mais complexa, pois além da aplicação da substituição, o algoritmo subtrai do contexto de variáveis V todos os elementos do domínio da substituição `s1` gerados pela a primeira chamada recursiva.

Logo, é preciso que os termos `(apply_subst s1 r1)` `(apply_subst s1 r2)` sejam bem-formados em relação a `(minus (get_ctxt l) (dom s1))`[5]. Para provar que a segunda chamada recursiva segue a ordem lexicográfica é necessário realizar uma análise de caso na substituição `s1`: se ela for vazia, então os tipos não se alteram, caso contrário o número de elementos em V reduziu.

5 Aplica-se `substs_remove`.

Os enunciados de consistência e completude do algoritmo de unificação são:

- Consistência: a substituição gerada S é um unificador para os tipos fornecidos.
- Completude: o unificador S é o mais geral, então para qualquer unificador S' existe um unificador S'' tal que $S' = S \circ S''$.

Essas noções estão embutidas no tipo `unify_type` (ver Figura 14), utilizado na certificação do resultado do algoritmo de unificação. Como as substituições de tipos estão codificadas como listas, com uma operação de aplicação, a igualdade entre substituições somente pode acontecer por extensionalidade[6] e $S' = S \circ S''$ é declarado como:

```
forall v, apply_subst s' (var v) =
          apply_subst (compose_subst s s'') (var v).
```

A entrada inicial do algoritmo é um dado do tipo `constraints`, tal que a condição de boa-formação é satisfeita. O tipo `sum` é utilizado para representar o sucesso ou o fracasso do algoritmo. Esse último definido pelo tipo `UnifyFailure`, o qual guarda a informação do erro ocorrido. Em caso de sucesso, o retorno do algoritmo é uma substituição com a certificação de consistência e completude. Os tipos `new_tv_ty` e `new_tv_subst` estão relacionados com a prova de completude do algoritmo W, e basicamente garantem que a unificação não cria novas variáveis de tipos.

Na unificação original foi utilizado o tipo `sumor: Type -> Prop -> Type` ao invés do `sum:Type -> Type -> Type`, porém a segunda componente do `sumor`, utilizada para armazenar a informação do erro, está em `Prop` e seria eliminada na extração do código. A informação do erro é um aspecto computacionalmente relevante e não deve ser eliminada na extração. O tipo `sum`, com ambas as componentes em `Type`, é mais adequado para isso.

Mais uma vez, o caso difícil de provar é o recursivo. Precisa-se mostrar a existência de uma substituição `s''`, tal que para a composição das substituições `s1` e `s2`, geradas nas chamadas recursivas, a condição de completude é satisfeita. Ou seja,

```
exists s'', forall v,
  forall v, apply_subst s' (var v) =
           apply_subst (compose_subst (compose_subst s1 s2) s'') (var v),
```

[6] Duas substituições de tipo são iguais se na aplicação têm os mesmos resultados para os mesmos argumentos.

Figura 14 – Tipo para a certificação do algoritmo de unificação.

```
Definition unify_type (c : constraints) := wf_constraints c ->
  { s | unifier (fst (get_tys c)) (snd (get_tys c)) s /\ wf_subst (get_ctxt c) s /\
    (forall st, (new_tv_ty (fst (get_tys c)) st /\ new_tv_ty (snd (get_tys c)) st) ->
    new_tv_subst s st) /\ forall s', unifier (fst (get_tys c)) (snd (get_tys c)) s' ->
    exists s'', forall v,
      apply_subst s' (var v) = apply_subst (compose_subst s s'') (var v)}
  +  (UnifyFailure (fst (get_tys c)) (snd (get_tys c))).
```

Fonte: o autor.

onde `s'` é um unificador para os tipos (`arrow`) a serem unificados. Essa prova segue diretamente da reescrita das hipóteses providas das duas chamadas recursivas.

4 CERTIFICAÇÃO DO ALGORTIMO W MONÁDICO

Esta é a parte central deste trabalho, cuja inspiração principal é a formalização do algoritmo W realizada por (DUBOIS; MÉNISSIER-MORAIN, 1999), a qual será referenciada como o *trabalho original*. Ainda que existam algumas diferenças de design desta formalização em relação ao trabalho original, muitos aspectos das provas são iguais, logo a maior parte do foco deste capítulo será nas diferenças relevantes. De qualquer forma, este texto é auto-contido e não é preciso consultar o artigo do trabalho original.

A implementação deste trabalho não é uma modificação do código do trabalho original, pois o código fornecido por Dubois é para Coq 6.1 o qual pode ser executado apenas parcialmente em versões mais recentes do Coq. Consequentemente, outra contribuição deste trabalho é fornecer uma revitalização da certificação do algoritmo W em Coq. O código deste trabalho foi testado na versão 9.1 do Coq.

4.1 FORMALIZAÇÃO DE DAMAS-MILNER E DO ALGORITMO W

A linguagem dos termos é dada pelo tipo indutivo apresentado na Figura 15, como uma tradicional linguagem no estilo ML. Escolhe-se não incluir termos para programas recursivos (`rec`) uma vez que as provas de consistência e completude para esse caso são as mesmos da abstração lambda (`lam_t`).

Figura 15 – Termos da linguagem estilo ML.

```
Inductive term : Set :=
| var_t   : id -> term
| app_t   : term -> term -> term
| let_t   : id -> term -> term -> term
| lam_t   : id -> term -> term
| const_t : id -> term.
```

Fonte: o autor.

Tipos polimórficos, ou *schemes*, estão definidos pelo tipo indutivo da Figura 16 e são separados dos tipos simples (monomórficos) utilizados na unificação (rever Figura 10). Essa separação é necessária para garantir que em determinadas situações o tipo não tem variáveis quantificadas, visto que isso facilita algumas formulações e provas[1]. Ambos os tipos são quase os mesmos, exceto pelo construtor `sc_gen` que representa variáveis de tipo quantificadas, as quais no algoritmo são obtidas pela ge-

[1] Um exemplo claro disso será dado na definição das regras de tipagem.

neralização de variáveis de tipo `var`. A aplicação da substituição de tipos para *schemes* (`schm`), chamada de `apply_subst_schm`, é como a para tipos monomórficos da Figura 10, porém trata-se `sc_gen` da mesma forma que constantes.

Figura 16 – Definição de tipos polimórficos (*schemes*).

```
Inductive schm : Set :=
| sc_var : id -> schm
| sc_con : id -> schm
| sc_gen : id -> schm
| sc_arrow : schm -> schm -> schm.
```

Fonte: o autor.

O contexto de tipos, denotado por Γ como na Seção 2.1, é apenas uma lista de pares de identificadores e *schemes* (`list (id * schm)`). Trivialmente, funções como `apply_subst_schm` são estendidas para contextos. Adicionar um elemento no contexto é apenas o operador `::` e a função `in_ctx` decide se um identificador `x` está presente em um contexto Γ.

O papel principal dos *schemes* é exercido na quantificação e na instanciação, os quais são representados por suas respectivas regras gen e $spec$ no sistema Damas-Milner (rever a Figura 2). Assim como feito em outras provas mecanizadas de algoritmos de inferência[2], as regras de tipagem aqui são dadas em um estilo *syntax-directed* (ver Figura 17), a fim de que as árvores de provas tenham um único formato para cada termo lambda. Dessa forma, as regras de quantificação e instanciação estão embutidas, respectivamente, nas regras para variáveis e *lets*. A equivalência desta versão com a original também foi formalizada em Coq por (DUBOIS, 1998).

Figura 17 – Versão *syntax-direct* das regras de tipagem como uma relação indutiva.

```
Inductive has_type : ctx -> term -> ty -> Prop :=
| const_ht : forall x G, has_type G (const_t x) (con x)
| var_ht : forall x G sigma tau, in_ctx x G = Some sigma ->
                                  is_schm_instance tau sigma -> has_type G (var_t x) tau
| lam_ht : forall x G tau tau' e, has_type ((x, ty_to_schm tau) :: G) e tau' ->
                                   has_type G (lam_t x e) (arrow tau tau')
| app_ht : forall G tau tau' l rho, has_type G l (arrow tau tau') ->
                                     has_type G rho tau -> has_type G (app_t l rho) tau'
| let_ht : forall G x e e' tau tau', has_type G e tau ->
                                      has_type ((x, gen_ty tau G) :: G) e' tau' ->
                                      has_type G (let_t x e e') tau'.
```

Fonte: o autor.

Esta formalização do algoritmo W utiliza tipos que são certificações de consistência e completude, as quais são diretamente expressas na pós-condição da HESM.

2 Por exemplo, (DUBOIS; MÉNISSIER-MORAIN, 1999; NARASCHEWSKI; NIPKOW, 1999)

Essa escolha de design tem duas principais vantagens em relação ao trabalho original: tipos como certificações podem reduzir trabalho nas provas[3] e a mônada permite a escrita do algoritmo W em Coq de um jeito mais usual (ver Figura 18)

A instância da HESM utilizada é chamada `Infer`, com o tipo `id` como o tipo do estado da contagem das variáveis já utilizadas e o tipo `InferFailure` para registar todos os possíveis erros de inferência:

```
Definition Infer := @HoareState id InferFailure.
```

Assim, com esse método de programação com tipos dependentes, não é necessário enunciar lemas relacionados aos resultados do algoritmo W e às suas subrotinas. Por exemplo, a função de busca para contextos, chamada de `look_dep` na implementação, retorna uma prova de `in_ctx` (caso esteja presente) ou levanta uma exceção na HESM, logo não é necessário enunciar um lema relacionando `in_ctx` e o resultado de `look_dep`.

Para que algoritmo de unificação fosse utilizado pelo algoritmo W foi necessário escrever algumas funções para servir de interface. A primeira interface fornece a prova inicial `wf_constraints`[4]. Isso é feito computando uma lista com todas as variáveis dos tipos a serem unificados e a prova que essa lista satisfaz a relação `wf_constraints` para esses tipos. A segunda interface é apenas levantar (*lift*) o algoritmo para a HESM. A função `unify` utilizada no algoritmo W tem uma assinatura com tipos dependentes, a qual fornece todas as informações necessárias para as provas de consistência e completude do W.

4.1.1 Instanciação de Tipos

Os tipos *schemes* podem ser instanciados para tipos monomórficos por meio de uma substituição de tipos especial `apply_inst_subst`. Ela somente altera os valores das variáveis de tipos quantificadas (aquelas no construtor `sc_gen`). Define-se a noção de instância de *scheme*, presente na regra sobre variáveis na Figura 17, por:

```
Definition is_schm_instance (tau:ty) (sigma:schm) :=
  exists i_s: inst_subst, (apply_inst_subst i_s sigma)
                          = (Some tau).
```

O tipo `inst_subst` é apenas uma lista de tipos monomórficos (`list ty`), ao invés de uma lista de associatividade. Na operação de substituição para instanciação de

[3] Esse aspecto é discutido na Seção 4.4.
[4] A função com tipos dependes requer como argumento uma prova sobre os tipos a serem unificados são bem-formados em relação ao conjunto de variáveis utilizado como critérios de parada.

Figura 18 – Algoritmo W monádico em Coq.

```
Program Fixpoint W (e:term) (G:ctx) {struct e} :
  Infer (fun i => new_tv_ctx G i) (ty * substitution)
            (fun i x f => i <= f /\ new_tv_subst (snd x) f /\ new_tv_ty (fst x) f /\
            new_tv_ctx (apply_subst_ctx (snd x) G) f /\
            has_type (apply_subst_ctx ((snd x)) G) e (fst x) /\
            completeness e G (fst x) ((snd x)) i) :=
  match e with
  | const_t x => ret ((con x), nil)
  | var_t x => sigma <- look_dep x G ;
               tau_iss <- schm_inst_dep sigma ;
               ret ((fst tau_iss), nil)
  | lam_t x e' => alpha <- fresh ;
                  G' <- @addFreshCtx G x alpha ;
                  tau_s <- W e' G' ;
                  ret (arrow (apply_subst (snd tau_s) (var alpha)) (fst tau_s),
                       (snd tau_s))
  | app_t l r => tau1_s1 <- W l G ;
                 tau2_s2 <- W r (apply_subst_ctx (snd tau1_s1) G) ;
                 alpha <- fresh ;
                 s <- unify (apply_subst (snd tau2_s2) (fst tau1_s1))
                            (arrow (fst tau2_s2) (var alpha)) ;
                 ret (apply_subst s (var alpha),
                      compose_subst  (snd tau1_s1) (compose_subst (snd tau2_s2) s))
  | let_t x e1 e2  => tau1_s1 <- W e1 G ;
                      tau2_s2 <- W e2 ((x, gen_ty (fst tau1_s1)
                                 (apply_subst_ctx (snd tau1_s1) G)) ::
                                 (apply_subst_ctx (snd tau1_s1) G)) ;
                      ret (fst tau2_s2, compose_subst (snd tau1_s1) (snd tau2_s2))
  end.
```

Fonte: o autor.

schemes (`apply_inst_subst`), o número n na variável quantificada a ser substituída é associada com o tipo encontrado na n-ésima posição em `inst_subst`. Consequentemente, o tamanho da lista (`inst_subst`) deve ser ao menos o valor do maior `gen` no dado *scheme*, ou então, a operação `apply_inst_subst` retornará um erro.

A implementação dessa operação é dada na Figura 19. Os casos interessantes são o `sc_gen`, que utiliza a função `nth_error` para encontrar o correspondente elemento em `inst_subst`, e o `sc_arrow`, o qual primeiro executa a aplicação no lado esquerdo e, se for bem-sucedida, executa no lado direito.

4.1.2 Generalização de Tipos

A generalização de tipos é o processo de quantificar variáveis de tipos livres em um tipo monomórfico τ, as quais não ocorrem livres no contexto Γ considerado, e consequentemente um novo *scheme* é criado. Como mencionado anteriormente, lidar com variáveis ligadas é uma dificuldade recorrente na formalização de aspectos de linguagens de programação, que geralmente é resolvida com índices de *de Bruijn*.

Figura 19 – Operação de substituição para instanciação de *schemes*.

```
Fixpoint apply_inst_subst (is_s:inst_subst) (sigma:schm) : option ty:=
  match sigma with
  | (sc_con c) => (Some (con c))
  | (sc_var v) => (Some (var v))
  | (sc_gen x) => match (nth_error is_s x) with
                   | None => None
                   | (Some t) => (Some t)
                 end
  | (sc_arrow ts1 ts2) => match (apply_inst_subst is_s ts1) with
                          | None => None
                          | (Some t1) =>
                            match (apply_inst_subst is_s ts2) with
                             | None => None
                             | (Some t2) => (Some (arrow t1 t2))
                            end
                         end
end.
```

Fonte: o autor.

Como exemplo motivacional sugerido por Dubois, considere que α e β são duas variáveis de tipo sintaticamente distintas e que não ocorrem livres no contexto considerado. Então, os tipos $\alpha \rightarrow \alpha$ and $\beta \rightarrow \beta$ são quantificados, respectivamente, para $\forall\alpha, \alpha \rightarrow \alpha$ e $\forall\beta, \beta \rightarrow \beta$. Esses *schemes* são sintaticamente diferentes, mas representam exatamente o mesmo conjunto de tipos. Essa ambiguidade adiciona complexidade na certificação do algoritmo.

Então, a fim de evitar lidar com equivalência alfa entre *schemes*, a conversão de um `var n` em `sc_gen m` deve seguir uma ordem numérica particular, o que fornece significado para a formulação dos índices de *de Bruijn* para variáveis de tipo (TAN; OWENS; KUMAR, 2015). As variáveis de tipo em um dado τ (`ty`) são quantificadas linearmente: se `var n` é a m-ésima variável de tipo descoberta quando τ é lido da esquerda para a direita, então essa é convertida em `sc_gen m`. Por fim, a equivalência de *schemes* é reduzida a igualdade sintática.

A implementação da quantificação é dada na Figura 20, separada em duas funções: `gen_ty_aux` e `gen_ty`. A primeira função recebe o tipo `tau` a ser quantificado e uma lista `l` de identificadores que já foram quantificados. Nessa função, para o caso de uma variável `var i` é verificado se `i` ocorre livre no contexto `G`. Caso sim, a variável não é quantificada. Caso não, então procura-se a posição[5] de `i` na lista de variáveis já quantificadas, a fim de evitar a quantificação de uma mesma variável em diferentes `sc_gen`. Se `i` não foi quantificado ainda, então a sua variável quantificada `sc_gen` tem o valor do tamanho da lista `l` e adiciona-se `i` em `l`.

5 A função `index_list_id` retorna o índice de um identificador se esse estiver presente na lista.

Figura 20 – Operação de quantificação de tipos.

```
Fixpoint gen_ty_aux (tau:ty) (G:ctx) (l:list id) : schm * list id :=
  match tau with
  | var i => if in_list_id i (FV_ctx G) then (sc_var i, l) else
             match index_list_id i l with
             | None => (sc_gen (List.length l), (l ++ i::nil))
             | Some j => (sc_gen j, l)
             end
  | con i => (sc_con i, l)
  | arrow tau' tau'' => match gen_ty_aux tau' G l with
                        | (sc_tau, l') => match gen_ty_aux tau'' G l' with
                                          | (sc_tau', l'') =>
                                            (sc_arrow sc_tau  sc_tau', l'')
                                          end
                        end
  end.

Definition gen_ty (tau:ty) (G:ctx) :=
  @fst schm (list id) (gen_ty_aux tau G nil).
```

Fonte: o autor.

4.2 CONSISTÊNCIA DO ALGORITMO W

O teorema da consistência é enunciado por meio da relação `has_type` na pós-condição de `Infer` na Figura 18. Conforme a definição de consistência para o algoritmo W da Seção 2.1, aplica-se a substituição resultante da inferência no contexto inicial `G`.

A extensão `Program` gera algumas obrigações de prova para cada caso de `e` no algoritmo W, as quais contêm a sequência de conjunções da pós-condição. Onde há chamada recursiva, `Program` gera uma obrigação de prova sobre a pré-condição da mesma. Devido aos tipos dependentes das funções utilizadas, em geral os casos da prova de consistência seguem diretamente das informações já no contexto do Coq e sem a necessidade de lemas auxiliares, ou seja:

- Caso de constante: trivial.
- Caso de variável: a substituição gerada é vazia, então apenas demostra-se que a instância computada de fato respeita a definição `is_schm_instance`, o que é trivial pois essa informação foi fornecida juntamente da instância.
- Caso de abstração lambda: trivial.
- Caso de aplicação: a substituição gerada é provida da composição das substituições das duas chamadas recursivas ($s1$ e $s2$) e da unificação (s). A prova é feita pelo uso do lema da estabilidade da tipagem sobre a substituição[6] e por

[6] Definido numa próxima subseção.

reescrita com a hipótese provida pela unificação.

O caso *let* é o mais trabalhoso, pois requer vários lemas e definições auxiliares, incluindo sobre listas disjuntas[7], sub-listas e substituições de renomeação. Os detalhes da prova deste caso são deixados para uma subseção posterior a essas definições auxiliares.

4.2.1 Substituição de Renomeação

Durante o caso *let* da prova de consistência é necessário lidar com uma forma de comutação entre aplicação de uma substituição de tipos e o processo de quantificação, o qual somente pode acontecer se não houver um conflito de variáveis. Para evitar esse conflito, utiliza-se um novo tipo de substituição que serve para renomear essas variáveis. Essa nova substituição possui as seguintes características:

- Cada variável no domínio (`id`) mapeia para uma variável (`id`).
- O domínio e a imagem da substituição são disjuntos.
- Duas distintas variáveis no domínio têm diferentes imagens.

Essa nova substituição tem o seu próprio tipo `ren_subst`, implementado como uma lista de associatividade entre identificadores (`list (id * id)`). A sua implementação em Coq é dada pelo predicado indutivo da Figura 21. Uma substituição de renomeação pode ser convertida para uma substituição simples por meio da função `rename_to_subst`, assim é possível reutilizar alguns lemas e definições.

Figura 21 – Definição da substituição de renomeação.

```
Inductive is_rename_subst : ren_subst ->  Prop :=
| is_rename_intro : forall r, (are_disjoints (dom_ren r) (img_ren r)) ->
                      (forall (x y: id), (in_list_id x (dom_ren r)) = true ->
                        (in_list_id y (dom_ren r)) = true ->
                        x <> y -> (apply_ren_subst r x) <> (apply_ren_subst r y)) ->
                        is_rename_subst r.
```

Fonte: o autor.

A prova do caso *let*[8] requer a demostração da existência de uma substituição de renomeação em especial, cujo domínio é uma lista de variáveis `l` e que não mapeia para as listas de variáveis `l1` e `l2`. O predicado da Figura 22 formaliza essa especificação.

[7] Cujo predicado chama-se `are_disjoint`
[8] Para a prova de consistência do algoritmo W e do lema da estabilidade da tipagem sobre a substituição

Figura 22 – Predicado para a computação da substituição de renomeação especial.

```
Inductive renaming_of_not_concerned_with:
  (ren_subst) -> (list id) -> (list id) -> (list id) -> Prop :=
| renaming_of_not_concerned_with_intro : forall (r:ren_subst) (l l1 l2: (list id)),
    (is_rename_subst r) -> (dom_ren r) = l ->
    (are_disjoints l1 (img_ren r)) -> (are_disjoints l2 (img_ren r)) ->
    (renaming_of_not_concerned_with r l l1 l2).
```

Fonte: o autor.

A técnica utilizada no trabalho original para demonstrar o existencial:

```
Lemma exists_renaming_not_concerned_with: forall (l l1 l2: (list id)),
  {r:ren_subst | (renaming_of_not_concerned_with r l l1 l2)}.
```

é uma prova convencional. Porém, dado a natureza computacional desse lema, utilizou-se uma função com tipos dependentes como mostra a Figura 23. Essa função computa uma substituição de renomeação, a qual mapeia os elementos de `l` para `is` superiores aos identificadores presentes nas listas `l1` e `l2`.

Figura 23 – Computação da substituição de renomeação especial.

```
Definition compute_renaming : forall (i:id) (l l1 l2:list id),
  (forall x, in_list_id x l = true -> x < i) ->
  (forall x, in_list_id x l1 = true -> x < i) ->
  (forall x, in_list_id x l2 = true -> x < i) ->
  {rho : ren_subst | is_rename_subst rho /\ dom_ren rho = l /\
                   (forall y, in_list_id y (img_ren rho) = true -> i <= y) /\
                   (forall x, in_list_id x (dom_ren rho) = true -> x < i) /\
                   are_disjoints l1 (img_ren rho) /\ are_disjoints l2 (img_ren rho)}.
refine (fix compute_renaming i l l1 l2 p p1 p2 :
        {rho : ren_subst | is_rename_subst rho /\ dom_ren rho = l /\
                         (forall y, in_list_id y (img_ren rho) = true -> i <= y) /\
                         (forall x, in_list_id x (dom_ren rho) = true -> x < i) /\
                         are_disjoints l1 (img_ren rho) /\
                         are_disjoints l2 (img_ren rho)} :=
          match l as y return l = y -> _ with
          | nil => fun _ => exist _ nil _
          | p::l' => fun H => match (compute_renaming (S i) l' l1 l2 _ _ _) with
                            | exist _ rl H => exist _ ((p, i)::rl) _
                            end
          end _).
```

Fonte: o autor.

Diversos outros lemas sobre a substituição de renomeação são necessários, em particular alguns relacionados com o processo de generalização de tipos e com listas disjuntas. Por exemplo, o lema da Figura 24 garante que uma substituição de renomeação `rho` não altera o resultado da generalização sobre a condição de não mapear as variáveis a serem quantificadas (dada pela função `gen_ty_vars`) para as variáveis livres do contexto `G`[9].

Figura 24 – Lema relacionando a quantificação de variáveis e a renomeação especial.

```
Lemma gen_ty_renaming: forall (G:ctx) (rho:ren_subst) (tau:ty) (s:substitution),
    (renaming_of_not_concerned_with rho (gen_ty_vars tau G) (FV_ctx G) (FV_subst s))
    -> (gen_ty tau G) = (gen_ty (apply_subst (rename_to_subst rho) tau) G).
```

Fonte: o autor.

4.2.2 Tipagem é Estável Sobre a Substituição

Essa propriedade clássica de sistemas de tipos diz que se $\Gamma \vdash \texttt{e} : \tau$, então para qualquer substituição S deve ser o caso que $S\Gamma \vdash \texttt{e} : S\tau$, ou enunciado em Coq:

```
Lemma has_type_is_stable_under_substitution :
  forall (e:term) (s:substitution) (G:ctx) (tau:ty), has_type G e tau ->
  has_type (apply_subst_ctx s G) e (apply_subst s tau).
```

A prova segue por indução em `e` e, assim como na prova de consistência do algoritmo W, a maioria dos casos desse lema são resolvidos pelas informações já presentes no contexto do Coq. O caso difícil do lema também é o *let*, o qual requer a comutação de uma generalização de tipo (de algum `tau`) com uma substituição de tipo `s`. No entanto, isso somente pode acontecer sobre certa condição: a substituição `s` não pode estar relacionada com as variáveis a serem quantificadas `tau`. Logo, tem-se o lema:

```
Lemma gen_ty_in_subst_ctx : forall (G:ctx) (s:substitution) (tau:ty),
  (are_disjoints (FV_subst s) (gen_ty_vars tau G)) ->
  (apply_subst_schm s (gen_ty tau G)) =
  (gen_ty (apply_subst s tau) (apply_subst_ctx s G)).
```

Assim, a prova do caso *let* de `has_type_is_stable_under_substitution`, requer que, antes de aplicar a substituição `s` em `tau`, seja aplicado a `tau` uma substituição `rho` que satisfaz a condição para a comutação. A substituição `rho` renomeia as variáveis de tipo em `tau` que devem ser generalizadas em novas variáveis de tipos que não ocorrem na substituição `s` e não são livres em `G`[10]. Consequentemente a premissa de `gen_ty_in_subst_ctx` é satisfeita.

[9] As variáveis livres da substituição `s` são irrelevantes neste caso.
[10] Utiliza-se o lema `exists_renaming_not_concerned_with`.

4.2.3 Caso *let* da Prova de Consistência

Este caso compartilha algumas características com o caso *let* da estabilidade da tipagem sobre a substituição. A substituição retornada é dada pela composição das substituições das duas chamadas recursivas (denotada por $s_1 \circ s_2$.). Mais uma vez, a comutação do procedimento de generalização com a aplicação de uma substituição, no caso $s_1 \circ s_2$, é novamente um requerimento.

Portanto, aplica-se uma substituição de renomeação `rho` que satisfaz a condição para a comutação. Este caso, ainda, requer o uso de alguns outros lemas, como a estabilidade da tipagem sobre a substituição.

4.3 COMPLETUDE DO ALGORITMO W

As regras de Damas-Milner (rever Seção 2.1) permitem julgamentos de tipos da forma $\Gamma \vdash \mathtt{e} : \sigma$[11], porém as regras no estilo *syntax-directed* da Figura 17 limitam-se a julgamentos da forma $\Gamma \vdash \mathtt{e} : \tau$[12]. Isso não foi um problema na especificação da consistência do algoritmo W, mas para a completude isso sugere uma reformulação de seu enunciado. A definição original de completude é: dados Γ e $\mathtt{e}$, seja Γ' uma instância de Γ e σ um *scheme* tal que

$$\Gamma' \vdash \mathtt{e} : \sigma$$

então

1. $W(\Gamma, \mathtt{e})$ termina com sucesso.
2. Se $W(\Gamma, \mathtt{e}) = (\mathbb{S}, \tau)$ então, para alguma substituição $\mathbb{S}'$,

$$\mathbb{S}'\mathtt{gen}(\mathbb{S}\Gamma, \tau) > \sigma \text{ e } \Gamma' = \mathbb{S}'\mathbb{S}\Gamma.$$

Na versão *syntax-directed* esse enunciado é reformulado, já em Coq, para:

```
Definition completeness (e:term) (G:ctx) (tau:ty) (s:substitution) (st:id) :=
  forall (tau':ty) (phi:substitution),
  has_type (apply_subst_ctx phi G) e tau' ->
  exists s', tau' = apply_subst s' tau /\
  (forall x:id, x < st ->
              apply_subst phi (var x) = apply_subst s' (apply_subst s (var x))).
```

onde, conforme a Figura 18, os argumentos fornecidos são respectivamente o termo e o contexto inicial fornecidos ao algoritmo W, o tipo e a substituição retornados pelo o mesmo, e o valor inicial do estado da HESM.

11 σ é um *scheme*
12 τ é um tipo monomórfico

A igualdade $\Gamma' = \mathbb{S}'\mathbb{S}\Gamma$ está implicitamente enunciada por

```
apply_subst phi (var x) = apply_subst s' (apply_subst s (var x)),
```

para somente valores `x` utilizados e que podem estar no contexto. Essa formulação é mais conveniente pois segue diretamente do uso de unificador mais geral no caso da aplicação.

A noção que o algoritmo W infere o tipo mais geral está implícita em

```
exists s', tau' = apply_subst s' tau,
```

mas feita explícita em $\mathbb{S}'\mathtt{gen}(\mathbb{S}\Gamma, \tau) \geq \sigma$. A relação entre *schemes* de mais geral ($\geq$) também foi formalizada em Coq[13], pois é parte do caso *let* da prova de completude.

No trabalho original, muitos passos computacionais foram necessários para a prova de completude, visto que a prova aconteceu por indução no termo `e` em um lema a respeito do resultado do algoritmo. Porém, neste trabalho, o uso de tipos dependentes evitou tais computações durante a prova. Para cada caso da prova de completude, calculados por `Program`, a principal complicação é computar a substituição `s'` da definição acima. Novamente, deixa-se o caso *let* para uma próxima subseção. Considerando os nomes `tau`, `tau'`, `phi s'` da definição `completeness` dada acima, então a prova do completude enunciada na pós-condição do algoritmo W foi:

- Caso de constante: trivial.
- Caso de variável: considere que `tau'` é uma instância de algum `sigma'` por meio da instanciação `is_s':inst_subst` e `tau` é uma instância de algum `sigma` por meio da da instanciação `is_s:inst_subst`[14]. Computa-se uma substituição que mapeia os identificadores de `is_s'` em em variáveis com os identificadores de `is_s` e, na mesma, concatena-se (`++`) a substituição `phi` para também substituir as variáveis monomórficas da tipagem de `tau'`. Com essa substituição, aplicada em `tau`, ambas as equações podem ser verificadas por meio de lemas auxiliares.
- Caso abstração lambda: A substituição `s'` é a própria provida da completude (hipótese de indução) da chamada recursiva.
- Caso aplicação: Neste caso, com duas chamadas recursivas, tem-se que o tipo inferido para o termo esquerdo é `tauLR`, o qual é unificado com `arrow tauL alpha`, onde `tauL` é o tipo inferido para o termo direito e `alpha` é uma variável *fresh*. A primeira equação do objetivo espera que `s'` satisfaça

[13] Discutida numa próxima subseção.
[14] Essas informações, que podem ser derivadas das hipóteses de indução, já estão presentes no contexto devido ao uso de tipos dependentes.

```
tau' = apply_subst s' (apply_subst mu alpha)
```

onde `mu` é a substituição unificadora. Para provar isso, considera-se a hipótese sobre `mu` ser o unificador mais geral

```
forall s0,
 apply_subst s0 (apply_subst s2 tauLR) = apply_subst s0 (arrow tauL alpha) ->
 exists s'', forall tau,
   apply_subst s0 tau = apply_subst (compose_subst mu s'') tau,
```

e toma-se `s0` como `((alpha, tau_r)::psi2`, onde `psi2` é a substituição provida pela completude da segunda chamada recursiva. Dessa forma, toma-se `s'` como `s''` e ambas equações são resolvidas por meio de reescritas com a hipótese do unificador mais geral.

Os casos de variável, aplicação e expressão *let* dependem da criação de novas variáveis de tipo. O fato das instanciações `is_s` e `is_s'` serem listas de variáveis *fresh* é fundamental, por exemplo.

Além de novas variáveis de tipo, o caso de expressão *let* também precisa do desenvolvimento de definições e lemas auxiliares sobre a relação mais geral (definida como $\geq$ na Seção 2, mas reinterpretada de forma mais estrita na Subseção 4.3.2), visto que para a completude é relevante que o processo generalização do algoritmo W com o contexto `G` resulte num tipo mais geral que no contexto `apply_subst_ctx phi G`.

4.3.1 Novas Variáveis de Tipo

Nota-se que para qualquer implementação do algoritmo W funcionar corretamente, a habilidade de produzir novas variáveis de tipos é essencial, assim como para a prova de completude. Esse aspecto é tomado como garantido em uma prova informal com papel e caneta, mas em um assistente de provas deve ser explicitamente formalizado.

Como nesta implementação as variáveis de tipos não são nominais (GABBAY; PITTS, 2002), são apenas números naturais, então é suficiente mostrar que o *atual estado* na mônada é maior que todos os números das variáveis de livres no contexto atual. Isso é exatamente expresso na pré-condição na mônada `Infer` na Figura 18. Consequentemente, sempre é possível produzir uma variável de tipo nova ao incrementar o estado na mônada.

Isso traz uma específica noção formal de novas variáveis de tipo para tipos monomórficos (`new_tv_ty`), para *schemes* (`new_tv_schm`), para contextos (`new_tv_ctx`) e para substituições (`new_tv_subst`). Para qualquer estrutura r, uma variável de tipo `i` é considerada nova em r se for maior que todas as variáveis de tipo livres ocorrendo nesta. Por exemplo, a definição indutiva de nova variável de tipos para *schemes* na Figura 25. As demais definições são análogas.

Figura 25 – Novas variáveis de tipos para *schemes*.

```
Inductive new_tv_schm : schm -> id -> Prop :=
  | new_tv_sc_con : forall i i':id,
                           new_tv_schm (sc_con i') i
  | new_tv_sc_gen : forall i i':id,
                           new_tv_schm (sc_gen i') i
  | new_tv_sc_var : forall i i':id, i < i' ->
                            new_tv_schm (sc_var i) i'
  | new_tv_sc_arrow : forall (tau tau':schm) (i:id),
                      new_tv_schm tau i ->
                      new_tv_schm tau' i ->
                      new_tv_schm (sc_arrow tau tau') i.
```

Fonte: o autor.

Vários lemas sobre essas quatro definições são necessários para realizar a prova de completude. Alguns lemas relacionando os resultados do algoritmo W com a noção de novas variáveis de tipos estão enunciados com `new_tv_ty`, `new_tv_subst` e `new_tv_ctx` na pós-condição. Além disso, o tipo `unify_type` garante que o algoritmo de unificação, da Figura 11, não cria novas variáveis de tipo na substituição unificadora `s` ao enunciar que para quaisquer tipos `tau1` e `tau2` unificados, tem-se que:

```
(new_tv_ty tau1 i /\ new_tv_ty tau2 i) ->
                          new_tv_subst s i.
```

4.3.2 Relação Mais Geral

A relação mais geral, definida na Subseção 2.1 como $\geq$ pode ser reinterpretada de forma mais estrita por meio da definição de instância (`is_schm_instance`) como: se um *scheme* σ_1 é mais geral que outro σ_2, então toda instância de σ_2 também é uma instância de σ_1. Em Coq, isso se traduz facilmente para:

```
Inductive more_general : schm -> schm -> Prop :=
    | more_general_intro : forall sigma1 sigma2:schm,
      (forall tau:ty, is_schm_instance tau sigma2 ->
                      is_schm_instance tau sigma1) ->
                      more_general sigma1 sigma2.
```

o qual se estende para contextos de forma restritiva, forçando que todos os mesmos identificadores nos dois contextos dados Γ_1 e Γ_2 estão na mesma posição. Logo, Γ_1

é mais geral que Γ_2 se o *scheme* do n-ésimo identificador i de Γ_1 é mais geral que o *scheme* do n-ésimo identificador i de Γ_2. Dois contextos vazios são mutualmente mais gerais um ao outro. Ou em Coq,

```
Inductive more_general_ctx : ctx -> ctx -> Prop :=
| more_general_ctx_nil : more_general_ctx nil nil
| more_general_ctx_cons : forall (G1 G2:ctx) (i:id) (sigma1 sigma2:schm),
    more_general_ctx G1 G2 -> more_general sigma1 sigma2 ->
    more_general_ctx ((i, sigma1)::G1) ((i, sigma2)::G2).
```

Vários lemas sobre essas relações são necessários para a prova de completude, em especial dois são complexos de provar e são utilizados diretamente no caso *let* da completude. O primeiro é outra propriedade clássica de sistemas de tipos e é sobre “tipagem em um contexto mais geral”: se é possível construir uma prova de $\Gamma_2 \vdash \texttt{e} : \tau$, então também é possível construir uma prova de $\Gamma_1 \vdash \texttt{e} : \tau$, onde Γ_1 é mais geral que Γ_2. A prova segue por indução em `e` e seu caso mais difícil é o *let*, que necessita de um lema sobre o processo de generalização em um contexto mais geral:

```
Lemma more_general_gen_ty : forall (G1 G2:ctx) (t:ty),
    more_general_ctx G1 G2 -> more_general (gen_ty t G1) (gen_ty t G2).
```

O segundo lema diz que aplicar uma substituição após o processo de generalização resulta em um *scheme* mais geral do que aplicar uma substituição antes da generalização:

```
Lemma more_general_gen_ty_before_apply_subst :
  forall (s:substitution) (G:ctx) (tau:ty),
   more_general (apply_subst_schm s (gen_ty tau G))
    (gen_ty (apply_subst s tau) (apply_subst_ctx s G)).
```

4.3.3 Caso *let* da Prova de Completude

A substituição `s'`, que precisa ser encontrada, é exatamente a fornecida pela propriedade de completude relacionada a segunda expressão `e2` (fornecida pela chamada recursiva). Assim, basta satisfazer a premissa da mesma, ou seja, a tipagem de `e2` com um contexto

```
apply_subst_ctx phi1 ((st0, gen_ty tau_e1 (apply_subst_ctx s1 G))::apply_subst_ctx s1 G)
```

onde `phi1` é a substituição `s'` referente a `e1`, e `s1` e `tau_e1` são referentes a inferência de `e1`. Esse objetivo é uma tipagem para `e2` com um contexto mais geral que a fornecida pelo objetivo inicial[15]. A prova é finalizada pelo uso dos dois lemas apresentados na subseção anterior. Em resumo, demonstra-se que a tipagem de `e2` feita pelo algoritmo W utiliza um contexto mais geral que qualquer outra tipagem.

4.4 ANÁLISE DO MÉTODO DE PROVA

Nesta seção pontua-se alguns detalhes sobre o método de prova utilizado, em especial como a HESM foi útil na prova de consistência e completude, e faz-se comparações com o trabalho original de (DUBOIS; MÉNISSIER-MORAIN, 1999). Uma vez que este trabalho utilizou métodos de automação nas provas, comparar o número de linhas com o trabalho original seria inadequado, portanto os aspectos da automação são ignorados nesta seção. Ao invés disso, compara-se quais lemas foram evitados e como certas formulações ficaram diferentes.

A verificação de programas em Coq pode ser feita de diferentes formas (PAULIN-MOHRING, 2012), em particular as técnicas mais comuns são:

- Escreve-se um programa P na linguagem *Gallina*, como se faria em qualquer linguagem de programação funcional. Posteriormente, define-se a certificação do programa P como um lema em Coq e prova-se o mesmo. Este método pode ser chamado de "certificação clássica" (CHLIPALA, 2013) e foi o método utilizado no trabalho original.
- Escreve-se um programa P como um termo t e a especificação como um tipo T, de forma que $t : T$ implica que o programa P está correto. Este método é conhecido como programação com tipos dependentes.

A HESM é um tipo dependente[16], portanto as vantagens da programação com tipos dependentes também se aplicam em seu uso. Isso é, a certificação é feita na mesma etapa da programação: o programa, cujo tipo é a sua especificação, e a sua prova são desenvolvidos simultaneamente. Uma desvantagem dessa técnica é que a escrita do programa frequentemente fica poluída com casamentos de padrão de tipos dependentes e elementos de provas. A extensão `Program` (SOZEAU, 2007), utilizada nessa prova, aprimora essa técnica ao permitir que elementos da prova sejam postergados como obrigações de provas e a escrita do programa seja realizada como se fosse com tipos simples.

O uso dessa extensão resultou em obrigações de provas correspondentes exatamente às pré-condições e pós-condições do algoritmo W. As informações presentes no contexto são similares as suposições feitas na prova com papel e caneta

[15] A definição `completeness` fornece hipóteses de tipagem.
[16] Visto que foi definido com o tipo `sig`.

do algoritmo W (DAMAS, 1984), incluindo as hipóteses de indução referentes às chamadas recursivas, as quais incluem as informações sobre a pós-condição de cada chamada recursiva. Portanto, no caso da aplicação (`app_t l r`) tem-se as hipóteses que os termos `l` e `r` têm as propriedades de consistência e completude. A certificação do trabalho original precisou de diversos lemas auxiliares para obter tais hipóteses de indução.

Para a prova de consistência, o trabalho original provou lemas que são hipóteses de indução do algoritmo W para cada um dos casos de termos. Por exemplo, para a aplicação (`app_t l r`) provou-se:

```
forall (l : term) (st1 st1' : id) (G1 : ctx) (tau1 : ty) (s1 : substitution),
W st1 G1 l = Some tau1 s1 st1' -> has_type (apply_subst_ctx s1 G1) l tau1 ->
forall (r : term) (st2 st2' : id) (G2 : ctx) (tau2 : ty) (s2 : substitution),
W st2 G2 r = Some tau2 s st' -> has_type (apply_subst_ctx s2 G2) r tau2 ->
forall (st3 st3' : id) (G3 : ctx) (tau3 : ty) (s3 : substitution),
W st3 G3 r = Some tau3 s st3' -> has_type (apply_subst_ctx s3 G) (appl_t l r) tau3.
```

Para provar essas hipóteses de indução foi necessário provar lemas sobre os resultados do algoritmo W para cada caso. Dubois chamou esses lemas de inversão (*inversion*) dos casos do algoritmo W. Por exemplo, no caso da inferência de uma abstração $\lambda x.x$ com tipo τ, então necessariamente τ é $\alpha \to \tau'$, para alguma variável *fresh* α e tipo τ'. Além disso, sabe-se que o algoritmo W foi bem-sucedido quando aplicado a e em um contexto Γ estendido com $x : \alpha$. Desconsiderando esses lemas de inversão, as provas dessas hipóteses de indução são essencialmente as mesmas que as provas de consistência deste trabalho.

O trabalho original utiliza a passagem explícita de um contador na função W para raciocinar sobre a geração de variáveis *fresh*, mas como consequência foram necessários lemas sobre o valor do mesmo. Por exemplo, o lema

```
forall (e : term) (st st' : id) (G : ctx) (tau : ty) (s : substitution),
W st G e = Some tau s st' -> st <= st'.
```

garante que o algoritmo W somente pode incrementar o contador ou mantê-lo igual. Lemas similares foram necessários para todas as outras funções auxiliares utilizadas no algoritmo, como a função de unificação.

De maneira similar à prova de consistência, o trabalho original também precisou de lemas que são hipóteses de indução para a prova de completude. A prova desses lemas requereu passos computacionais, visto que o lema diz que o resultado do algoritmo W é um tipo τ, tal que para qualquer outra tipagem τ' (do programa e considerado) existe uma substituição s de forma que $\tau' = s\tau$. Assim, foi necessário efetuar os passos computacionais dentro do lema a fim de obter o tipo τ. Tais passos computacionais foram completamente evitados na formulação com tipos dependentes deste trabalho.

4.5 EXTRAÇÃO DO ALGORITMO W CERTIFICADO

A extração do algoritmo W certificado é o produto final deste trabalho. Acredita-se que esta é a primeira certificação do algoritmo W totalmente executável. As demais certificações completas, como de CakeML (TAN; OWENS; KUMAR, 2015) e da extensão de ML (GARRIGUE, 2015) não seguem estritamente a definição original do algoritmo, uma vez que reformulam o mesmo para lidar com outros sistemas que não o Damas-Milner.

O processo de extração remove as provas da formalização e deixa somente termos computacionalmente relevantes. Portanto, é de grande importância somente utilizar o tipo `Prop` nos lugares sem relevância na computação. A Mônada de Exceção-Estado de Hoare implementada na Subseção 2.4.2 utiliza o tipo `sum` para armazenar a disjunção de sucesso e fracasso do algoritmo W. Esse tipo é definido por:

```
Inductive sum (A B:Type) : Type :=
| inl : A -> A + B
| inr : B -> A + B.
```

no qual ambas as componentes `A` e `B` estão em `Type` e, portanto, não são eliminadas no processo de extração.

Exemplos de termos a serem eliminados na extração são a pré-condição e a pós-condição. A pré-condição é essencial para garantir que o algoritmo somente seja executado se o estado na mônada for novo em relação ao contexto dado[17]. Dessa forma, os dados relacionados a falha do algoritmo são preservados, pois o tipo `InferFailure` tem valores em `Set` que são relacionados a informação da falha, por exemplo, se a inferência falhou pois tentou unificar duas constantes de tipo diferentes.

A Figura 26 é uma função que fornece o estado inicial para a HESM, e assim, permite que o algoritmo W seja executado apenas com o termo `e` e o contexto `G`. O retorno é extraído da HESM e projetado para o tipo `sum`, sem as informações da certificação e do estado. A função `computeInitialState` permite que isso sempre seja satisfeito, pois computa o estado inicial para o contexto dado e também fornece a prova que esse estado é novo em relação ao contexto.

---
[17] Um valor maior que todos no contexto.

Figura 26 – Função para a execução do algoritmo W.

```
Program Definition runW e G : sum (ty * substitution) InferFailure :=
  match W e G (computeInitialState G)  with
  | inl _ (a', _) => inl _ a'
  | inr _ er => inr _ er
  end.
```

Fonte: o autor.

O processo de extração, na sua forma mais simples, é bastante trivial. Basta avisar ao Coq a linguagem destino[18] e utilizar o comando de extração na função desejada, que Coq irá extrair recursivamente todas as definições necessárias e eliminar todos os termos em `Prop`.

Por outro lado, é possível mapear os tipos indutivos de Coq em tipos da linguagem destino. Para isso, deve-se dizer como os nomes e os construtores desses tipos serão mapeados para os respectivos nomes e construtores na linguagem destino. Também é possível mapear funções de Coq como constantes, por exemplo a função `plus` para o operador `(+)` em Haskell. Nota-se que é de extrema importância a certeza de que os mapeamentos são seguros, ou seja, que não comprometem o funcionamento esperado das funções extraídas. Em geral, deseja-se realizar essas traduções por questões de eficiência. Os inteiros nativos de Haskell/OCaml/Scheme são mais eficientes que os inteiros unários de Coq, por exemplo.

Nesta extração optou-se como linguagem destino Haskell. Decidiu-se mapear os tipos indutivos conforme a Figura 27 para evitar redundância com os tipos já presentes em Haskell. A representação de identificadores como números naturais (`nat`) foi mapeada para o tipo `Int`, pelo uso mais eficiente de memória.

Figura 27 – Mapeamentos dos tipos indutivos da extração.

```
Extract Inductive bool => "Prelude.Bool" ["Prelude.True" "Prelude.False"].
Extract Inductive sumbool => "Prelude.Bool" ["Prelude.True" "Prelude.False"].
Extract Inductive list => "[]" [ "[]" "(:)" ].
Extract Inductive prod => "(,)"  [ "(,)" ].
Extract Inductive sum => "Prelude.Either" ["Prelude.Left" "Prelude.Right"].
```

Fonte: o autor.

A fim de facilitar o uso do código extraído, implementou-se um parser com a biblioteca *Parsec*. Esse parser não é certificado, mas dada a sua simplicidade não se espera a presença de erros. Implementou-se, também, as instâncias da classe `Show` para os novos tipos criados pela a extração. Em particular, os identificadores são

[18] Haskell, OCaml ou Scheme.

impressos na tela como *strings*. A Figura 28 contém alguns exemplos da execução do código extraído.

Figura 28 – Exemplos de execução do algoritmo W certificado.

```
> let s = \x -> \y -> \z -> x z (y z) in s
> ((a -> (b -> c)) -> ((a -> b) -> (a -> c)))

> let fun = \f -> \x -> f x in fun 1
> Can't unify Int and (a -> b)

> \x -> x x
> Occurs check failure: a in (a -> b)

> let pair = \x -> \y -> \z -> z x y in
  let id = \x -> x in pair (id 1) (id id)
> ((Int -> ((a -> a) -> b)) -> b)
```

Fonte: o autor.

Nos (pequenos) casos de teste efetuados não se notou problemas de desempenho, as respostas foram retornadas em milésimos de segundo. O código extraído aparenta ser tão eficiente quanto uma implementação convencional em Haskell do algoritmo W, porém para confirmar isso um teste estatístico seria necessário. Uma vez que os identificadores foram mapeados para números inteiros (`Int`) eles não representam um gargalo no tempo de execução ou uma intensificação no uso de memória.

A função principal extraída, para inferência de tipos dos termos, é um código monádico apesar de não utilizar a função `bind` padrão do Haskell, visto que como mapear a HESM para uma monad transformer (`ExceptT` por exemplo) ainda precisa ser investigado. Ainda, seria interessante uma forma de preservar a notação *do* na extração, assim teria-se um código extraído mais legível.

5 CONCLUSÃO

Este trabalho apresentou uma certificação em Coq do algoritmo W para inferência de tipos no sistema Damas-Milner. Certificou-se em Coq que o algoritmo W é consistente e completo com as regras de Damas-Milner. Ainda, esta certificação é completa e independe de axiomas[1], portanto foi possível realizar a extração para Haskell de um código executável.

Ainda que certificações do algoritmo W já tenham sido apresentadas na literatura, nenhuma seguiu o estilo convencional de implementação com mônadas e as utilizou como artifício nas provas. A Mônada de Estado de Hoare, demostrada na Seção 2.4.1, permite a certificação de programas com efeitos de estado, com base no método de prova de Hoare-Floyd. Após modificar essa mônada para tratar exceções, foi possível que a implementação monádica do algoritmo W fosse verificada consistente e completa com o sistema Damas-Milner. Concluiu-se que essa mônada modificada é robusta o suficiente para tratar esse estudo de caso.

Para certificar o algoritmo W reutilizou-se uma certificação do algoritmo de unificação, a qual foi fornecida por (RIBEIRO; CAMARÃO, 2016). No entanto, algumas modificações não triviais foram necessárias para que a mesma pudesse ser utilizada na certificação do W. Em particular, modificou-se o algoritmo para unificar apenas dois tipos ao invés de uma lista de pares de tipos, o que implicou em modificações no critério de parada do mesmo, e modificou-se a definição de aplicação de substituição para estar em maior conformidade com a sua definição matemática e, assim, fornecer alguns lemas fundamentais para a prova de completude do algoritmo W.

Assim como no trabalho de (DUBOIS; MÉNISSIER-MORAIN, 1999) foi necessário definir três tipos distintos de substituição de tipos: a primeira ligando variáveis livres a tipos (`substitution`), a segunda ligando variáveis quantificadas a tipos (`inst_subst`) e a terceira que realiza renomeação de tipos (`ren_subst`). Essas distinções entre substituições foram necessárias exclusivamente por causa do polimorfismo. De fato, o processo de generalização (do polimorfismo) presente na regra `let_ht` é a principal origem de dificuldade nas provas de consistência e completude.

Toda a implementação deste trabalho em Coq conta com 2055 linhas de definições/especificações e 3804 linhas de provas[2], conforme o cálculo do *coqwc*. Ao todo foram 143 definições/especificações e 254 lemas/teoremas.

---

[1] Com exceção do axioma `eq_rect_eq` da biblioteca *Libtactics*.
[2] Excluindo a biblioteca Libtactics.

5.1 TRABALHOS FUTUROS

A pequena revisão da literatura sobre certificações de algoritmos de inferência de tipos na Seção 2.2 mostrou que ainda há poucos trabalhos nesse assunto. Em particular, ainda não há certificação completa do algoritmo de Wand, a qual poderia servir de fundamento para a certificação da inferência de tipos de uma linguagem de programação industrial, pois geralmente essas linguagens utilizam algoritmos *constraint-based*. A certificação de algoritmos de inferência para linguagens industriais é um tópico com muito trabalho a ser realizado.

Um aspecto importante deste trabalho foi modificar a Mônada de Estado de Hoare para suportar exceções. Nota-se, no entanto, que essa modificação é incompleta. A versão completa dessa mônada também permite verificar propriedades em caso de exceção e não apenas de sucesso. Por exemplo, caso o algoritmo de unificação retorne com sucesso, então a substituição retornada é provada ser um unificador e o mais geral. Mas caso a unificação falhe, o retorno poderia ser um dado com as informações do erro e uma prova de que os termos de entrada não são unificáveis.

Para um algoritmo de inferência de tipos, o sucesso é acompanhado de provas de consistência e completude, e a exceção é acompanhada de uma prova que o programa não é tipável no contexto considerado. Esse tipo de certificação é mais forte que a realizada neste trabalho, pois implica diretamente a decidibilidade do problema sendo resolvido pelo o algoritmo.

Uma proposta de Mônada de Exceção-Estado de Hoare completa é mostrada na Figura 29, cuja pós-condição `Post` passa incluir um tipo `e` a respeito da falha. Uma pós-condição ou garante uma propriedade do sucesso ou uma propriedade da falha. Por exemplo, a pós-condição do algoritmo W é

```
(fun i x f =>  match x with
               | inl (tau, s) => i <= f /\
                                 new_tv_subst s f /\ new_tv_ty tau f /\
                                 new_tv_ctx (apply_subst_ctx s G) f /\
                                 has_type (apply_subst_ctx s G) e tau /\
                                 completeness e G tau s i
               | inr r => ~ exists tau s, has_type (apply_subst s G) e tau
               end),
```

cujo caso de falha certifica que não existe tipo `tau` e substituição `s` aplicada ao contexto `G` que satisfaça a relação de tipagem para o programa `e`.

A pré-condição resultante da combinação das duas computações `c1` e `c2` (na função `bind`) não precisa garantir que a pós-condição da primeira computação im-

Figura 29 – A Mônada de Estado-Exceção de Hoare completa.

```
Variable st : Set.

Definition Pre : Type := st -> Prop.

Definition Post (a e : Type) : Type := st -> sum a e -> st -> Prop.

Program Definition HoareState  (e : Type) (pre : Pre) (a : Type)
  (post : Post a e) : Type :=
  forall i : {t : st | pre t}, {x : ((sum a e) * st) |
                                match x with
                                | (x, f) => post (proj1_sig i) x f
                                end}.

Program Definition bind : forall a b e P1 P2 Q1 Q2,
  (@HoareState e P1 a Q1) -> (forall (x : a), @HoareState e (P2 x) b (Q2 x)) ->
  @HoareState e (fun s1 => P1 s1 /\ forall x s2, match x as x' with
                                               | inl l => Q1 s1 x s2 -> P2 l s2
                                               | inr r => True
                                               end )
              b
              (fun s1 y s3 => exists x s2, match x, y as xy with
                                          | inl l, _ => Q1 s1 x s2 /\ Q2 l s2 y s3
                                          | inr r, inr _ => Q1 s1 x s2
                                          | inr r, inl _ => False
                                          end) :=
  fun a b e P1 P2 Q1 Q2 c1 c2 s1 => match c1 s1 as y with
                                    | (x, s2) => match x with
                                                 | inl l => c2 l s2
                                                 | inr r => (inr r, s2)
                                                 end
                                    end.
```

Fonte: o autor.

plique a pré-condição da segunda caso a primeira computação falhe, então toma-se apenas a pré-condição de `c1` nessa situação. Por isso o `True` no respectivo caso do casamento de padrão do resultado `x` da primeira computação.

A pós-condição da combinação depende do sucesso e da falha das duas computações, portanto faz-se `match x, y` para analisar os possíveis casos. Se a primeira computação for bem-sucedida, então deve-se considerar ambas pós-condições. Nota-se que a pós-condição da segunda computação depende de um dado de tipo `a` (e não `sum a e`), portanto somente faz sentido considerar a pós-condição da segunda computação no caso de sucesso da primeira. Consequentemente, caso a primeira computação falhe, então apenas considera-se a pós-condição da mesma. O último caso é um absurdo (`False`), pois nunca deve ser o caso da primeira computação falhar e a segunda ser bem-sucedida, visto que o comportamento de uma exceção é interromper próximas computações assim que a falha ocorre.

REFERÊNCIAS

AYDEMIR, B. et al. Engineering formal metatheory. In: **Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08**. [S.l.]: ACM Press, 2008. ISBN 9781595936899.

AYDEMIR, B. E. et al. Mechanized Metatheory for the Masses: The PoplMark Challenge. In: . [S.l.: s.n.], 2010.

BERTOT, Y.; CASTRAN, P. **Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642058809, 9783642058806.

BOVE, A. **Thesis for the Degree of Licentiate of Technology Programming in Martin-Lof Type Theory Uni cation A non-trivial Example**. Tese (Doutorado), 1999.

CHLIPALA, A. **Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant**. [S.l.]: The MIT Press, 2013. ISBN 0262026651, 9780262026659.

DAMAS, L. **Type assignment in programming languages**. Tese (Doutorado), 1985. PHD.

DAMAS, L.; MILNER, R. Principal type-schemes for functional programs. In: **Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 1982. (POPL '82), p. 207–212. ISBN 0-89791-065-6.

DAMAS, L. M. M. Type Assignment in Programming Languages. n. CST-33-85, 1984.

DIEHL, S. Write You a Haskell Building a modern functional compiler from first principles Write You a Haskell. 2015.

DUBOIS, C. Sûreté du typage de ML: Spécification et Preuve en Coq. **Journées Francophones des Langages Applicatifs**, n. 9, 1998.

DUBOIS, C.; MÉNISSIER-MORAIN, V. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. **Journal of Automated Reasoning**, v. 23, n. 3-4, p. 319–346, 1999. ISSN 01687433.

GABBAY, M. J.; PITTS, A. M. A new approach to abstract syntax with variable binding. **Formal Aspects of Computing**, v. 13, n. 3, p. 341–363, Jul 2002. ISSN 1433-299X.

GARRIGUE, J. A certified implementation of ML with structural polymorphism and recursive types. In: **Mathematical Structures in Computer Science**. [S.l.: s.n.], 2015. v. 25, n. 4, p. 867–891. ISSN 09601295.

GEUVERS, H. Proof assistants: History, ideas and future. **Sadhana**, v. 34, n. 1, p. 3–25, Feb 2009. ISSN 0973-7677.

GIBBONS, J.; HINZE, R. Just do it. **ACM SIGPLAN Notices**, v. 46, n. 9, p. 2, sep 2011. ISSN 03621340.

GONTHIER, G. The four colour theorem: Engineering of a formal proof. In: KAPUR, D. (Ed.). **Computer Mathematics**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 333–333. ISBN 978-3-540-87827-8.

GRABMÜLLER, M. **Algorithm W Step by Step**. [S.l.], 2006.

HINDLEY, R. The Principal Type-Scheme of an Object in Combinatory Logic. **Transactions of the American Mathematical Society**, American Mathematical Society, v. 146, p. 29–60, 1969. ISSN 00029947.

JONES, M. P. Typing Haskell in Haskell. **Haskell Workshop**, 1999.

JONES, S. L. P. **The Implementation of Functional Programming Languages**. [S.l.: s.n.], 1987.

Kiam Tan, Y. et al. The Verified CakeML Compiler Backend. **Journal of Functional Programming**, v. 29, 2019.

KOTHARI, S.; CALDWELL, J. A machine checked model of MGU axioms: applications of finite maps and functional induction. p. 1–13, 2009.

KOTHARI, S.; CALDWELL, J. A Machine Checked Model of Idempotent MGU Axioms For Lists of Equational Constraints. **Electronic Proceedings in Theoretical Computer Science**, v. 42, p. 24–38, 2010.

KOTHARI, S.; CALDWELL, J. L. On extending wand's type reconstruction algorithm to handle polymorphic let. **Fourth Conference on Computability in Europe**, 2008.

KOTHARI, S.; CALDWELL, J. L. **Toward a machine-certified correctness proof of Wand's type reconstruction algorithm**. 2009.

KUMAR, R.; NORRISH, M. (Nominal) unification by recursive descent with triangular substitutions. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.: s.n.], 2010. v. 6172 LNCS, p. 51–66. ISBN 3642140513. ISSN 03029743.

LEROY, X. et al. The compcert verified compiler. **Documentation and user's manual. INRIA Paris-Rocquencourt**, 2012.

MAHBOUBI, A.; TASSI, E. **Mathematical Components**. [S.l.: s.n.], 2018. 1–183 p.

MCBRIDE, C. First-order unification by structural recursion. **Journal of Functional Programming**, v. 13, n. 6, p. 1061–1075, 2003. ISSN 09567968.

MILNER, R. A theory of type polymorphism in programming. **Journal of Computer and System Sciences**, v. 17, p. 348–375, 1978.

MITCHELL, J. C. **Foundations of Programming Languages**. Cambridge, MA, USA: MIT Press, 1996. ISBN 0-262-13321-0.

NARASCHEWSKI, W.; NIPKOW, T. Type Inference Verified: Algorithm script W sign in Isabelle/HOL. **Journal of Automated Reasoning**, v. 23, n. 3-4, p. 299–318, 1999. ISSN 01687433.

O'CONNOR, R. Simplicity: A new language for blockchains. In: **Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security**. New York, NY, USA: ACM, 2017. (PLAS '17), p. 107–120. ISBN 978-1-4503-5099-0.

PARASKEVOPOULOU, Z. et al. Foundational property-based testing. In: URBAN, C.; ZHANG, X. (Ed.). **Interactive Theorem Proving**. Cham: Springer International Publishing, 2015. p. 325–343. ISBN 978-3-319-22102-1.

PAULIN-MOHRING, C. Introduction to the Coq proof-assistant for practical software verification. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.: s.n.], 2012. v. 7682 LNCS, p. 45–95. ISBN 9783642357459. ISSN 03029743.

PAULSON, L. C. Verifying the unification algorithm in LCF. **Science of Computer Programming**, v. 5, n. C, p. 143–169, 1984. ISSN 01676423.

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

PIERCE, B. C. et al. **Software Foundations**. [S.l.]: Electronic textbook, 2017.

REMY, D. et al. **Extension of ML Type System with a Sorted Equational Theory on Types**. 1992.

RIBEIRO, R.; CAMARÃO, C. A mechanized textbook proof of a type unification algorithm. In: CORNÉLIO, M.; ROSCOE, B. (Ed.). **Formal Methods: Foundations and Applications**. Cham: [s.n.], 2016. ISBN 978-3-319-29473-5.

ROBINSON, J. A. A machine-oriented logic based on the resolution principle. **J. ACM**, ACM, New York, NY, USA, v. 12, n. 1, p. 23–41, jan. 1965. ISSN 0004-5411.

SØRENSEN, M. H. B.; URZYCZYN, P. **Lectures on the Curry-Howard Isomorphism**. 1998.

SOZEAU, M. Subset coercions in coq. In: **Proceedings of the 2006 International Conference on Types for Proofs and Programs**. Berlin, Heidelberg: Springer-Verlag, 2007. (TYPES'06), p. 237–252. ISBN 3-540-74463-0, 978-3-540-74463-4.

SULZMANN, M.; ODERSKY, M.; WEHR, M. Type inference with constrained types. **Theory and Practice of Object Systems**, v. 5, n. 1, p. 35–55, 1999. ISSN 10743227.

SWIERSTRA, W. The Hoare State Monad. **Technology**, Springer, Berlin, Heidelberg, p. 440–451, 2009.

TAN, Y. K.; OWENS, S.; KUMAR, R. A verified type system for CakeML. In: . [S.l.: s.n.], 2015. p. 1–12. ISBN 9781450342735.

URBAN, C.; NIPKOW, T. **Nominal Verification of Algorithm W**. [S.l.], 2008.

WADLER, P. Comprehending monads. **Mathematical Structures in Computer Science**, v. 2, n. 4, p. 461–493, 1992. ISSN 0960-1295.

WAND, M. A Simple Algorithm and Proof for Type Inference. **Fundamenta Infomaticae**, v. 10, p. 115–122, 1987.

ZHAO, J. et al. Formalizing the llvm intermediate representation for verified program transformations. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 1, p. 427–440, jan. 2012. ISSN 0362-1340.