Universidade Estadual de Maringá

Centro de Tecnologia

Departamento de Informática

Implementação de um gerador de código Wasm para a linguagem de programação Gleam

Lucas Wolschick

TCC-2024

Maringá - Paraná

2024

Universidade Estadual de Maringá

Centro de Tecnologia

Departamento de Informática

Implementação de um gerador de código Wasm para a linguagem de programação Gleam

Lucas Wolschick

TCC-2024

Trabalho de Conclusão de Curso apresentado à Universidade Estadual de Maringá, como parte dos requisitos necessários à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Marco Aurélio Lopes Barbosa

Banca: Felippe Fernandes da Silva

Banca: Nilton Luiz Queiroz Junior

Maringá - Paraná

2024

Resumo

Gleam é uma linguagem de programação funcional, estaticamente tipada e com sintaxe simples e familiar. Gleam é versátil e pode ser usada em diversos contextos. A linguagem é compilada, e o compilador fornecido com a implementação da linguagem possui duas retaguardas (*back-ends*) geradoras de código: uma para JavaScript e uma para a máquina virtual Erlang (EVM/BEAM). A maioria dos usuários do Gleam usa a retaguarda Erlang, a qual é a mais antiga. Contudo, existe uma demanda dos usuários do Gleam por uma retaguarda WebAssembly (Wasm), uma linguagem projetada para ser alvo de compilação de linguagens de alto nível para a Web, leve e com propriedades que permitem implementações de alto desempenho. Neste trabalho, apresentamos uma implementação inicial de um gerador de código Wasm para o compilador da linguagem Gleam, que abrange um subconjunto considerável da linguagem, incluindo funções, tipos definidos pelo usuário, casamento de padrões, constantes e suporte a vários tipos da linguagem, o que permite o seu uso em algumas aplicações reais. Esperamos que o gerador de código Wasm desenvolvido nesse trabalho seja uma boa contribuição à comunidade de desenvolvedores Gleam e aos interessados no projeto e implementação de compiladores e de linguagens de programação.

Palavras-chave: WebAssembly, Gleam, geração de código, compiladores, linguagens de programação.

Abstract

Gleam is a functional, statically-typed programming language with a simple and familiar syntax. Gleam is versatile and can be used in various contexts. The language is compiled, and its compiler provides two code generation backends: one for JavaScript and one for the Erlang virtual machine (EVM/BEAM). Most Gleam users utilize the EVM backend, which is the oldest. However, there is user demand for a WebAssembly (Wasm) backend, a language designed as a compilation target for high-level languages for the Web, lightweight, and with properties enabling high-performance implementations. In this work, we present an initial implementation of a Wasm code generator for the Gleam language compiler, covering a significant subset of the language, including advanced features such as functions, user-defined types, pattern matching, constants, and support for various language types, allowing its use in several real-world applications. With this project, we aim to provide a high-quality Wasm code generator implementation to the Gleam programming community and to those interested in compiler and programming language design and implementation.

Keywords: WebAssembly, Gleam, code generation, compilers, programming languages.

Listagens de código

Lista de ilustrações

Lista de abreviaturas e siglas

API	Application Programming Interface (Interface de Programação de Aplicações)
AST	Abstract Syntax Tree (Árvore Sintática Abstrata)
EVM	Erlang Virtual Machine
GC	Garbage collection/collector (Coleção/Coletor de lixo)
GCC	GNU Compiler Collection
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
TDU	Tipo definido pelo usuário
UTF-8	Unicode Transformation Format — 8-bit
WASI	WebAssembly System Interface
Wasm	WebAssembly

Sumário

1 Introdução

Linguagens de programação são linguagens formais que descrevem computações para computadores e seres humanos (AHO et al., 2007). As linguagens de programação livram os seus usuários de ter que descrever programas diretamente em código de máquina, permitindo que os algoritmos e procedimentos sejam expressos de maneira abstraída dos detalhes subjacentes de implementação dos seus computadores.

As primeiras linguagens de programação surgiram quase que concomitantemente com os primeiros computadores de propósito geral. Em 1954, nove anos após a construção do ENIAC, surgiu a primeira linguagem de programação de alto nível compilada, o FORTRAN (SEBESTA, 2011), e nas décadas seguintes, linguagens como ALGOL (1958), LISP (1960) e Simula (1962) introduziram conceitos e paradigmas de programação presentes em linguagens usadas até hoje, como C (1972), C++ (1979), Python (1991) e Java (1996). A evolução e o surgimento de novos requisitos em vários domínios de *software* continuam incentivando inovações nessa área até hoje, resultando no desenvolvimento de linguagens como Go (2012), Rust (2015) e Zig (2016).

Na década de 1980, em meio a uma busca por ferramentas de desenvolvimento mais adequadas às suas necessidades, a empresa sueca de tecnologia e telecomunicações Ericsson desenvolveu a linguagem de programação Erlang[1], para ser utilizada na produção de *hardware* e *software* de telefonia (ARMSTRONG, 2007). A linguagem foi projetada para facilitar o desenvolvimento de aplicações distribuídas, altamente concorrentes e tolerantes a falhas, requisitos comuns na área de telecomunicações. Após alguns anos, com o desenvolvimento da Internet e da Web, houve um interesse renovado no ecossistema Erlang graças à sua adequação natural às demandas e necessidades das aplicações Web.

Erlang é funcional e possui uma sintaxe inspirada em Prolog, e é compilada e executada na máquina virtual BEAM, a qual implementa os recursos de confiabilidade e disponibilidade da linguagem. Conforme o ecossistema Erlang se popularizou, um desejo por linguagens de programação com sintaxes mais amigáveis e recursos mais flexíveis, mas que ainda aproveitassem as características da plataforma BEAM e o ecossistema de ferramentas e bibliotecas existentes, levou à criação de linguagens como Elixir (VALIM, 2017) e Gleam. A linguagem Gleam[2] é funcional e executa na BEAM, assim como Erlang e Elixir, mas diferentemente de ambas, é estaticamente tipada e possui uma sintaxe mais familiar a desenvolvedores vindo de linguagens da família C. Gleam possui um sistema de tipos Hindley-Milner (HINDLEY, 1969; MILNER, 1978), que é capaz de inferir automaticamente os tipos das declarações do programa: isso permite aos seus usuários escrever código com uma aparência mais simples e dinâmica, ao passo que retém a robustez de um sistema de tipos estático e forte. A sua comunidade de desenvolvedores é nova,

[1] Erlang. Disponível em: <https://erlang.org>
[2] Gleam. Disponível em: <https://gleam.run>

vibrante e ativa, e a implementação principal da linguagem Gleam é software livre, escrita em Rust[3]. Originalmente, a linguagem era apenas compilada para Erlang, mas um emissor de código JavaScript foi posteriormente adicionado ao compilador para que ela pudesse ser utilizada em mais espaços, como em navegadores *Web* e em plataformas de computação em nuvem.

1.1 Motivação

Devido a sua simplicidade e a suas características funcionais, Gleam pode ser usada como uma linguagem de propósito geral com uso em vários domínios, como no desenvolvimento de sistemas *Web*, de Internet das Coisas (IoT), inteligência artificial e desenvolvimento de jogos. Além desses casos de uso, a linguagem também pode se tornar uma boa opção para o ensino de programação em cursos de graduação. A ênfase em funções puras, as quais mapeiam entradas para saídas sem efeitos colaterais, assemelha o funcionamento da linguagem às intuições que os alunos desenvolvem a respeito de funções matemáticas vistas durante o ensino médio.

Contudo, alguns aspectos da linguagem Gleam dificultam a sua adoção em alguns destes contextos. A mais relevante para este trabalho é a sua dependência de um ambiente de execução Erlang ou JavaScript para que aplicações escritas na linguagem possam ser executadas. Esta restrição dificulta o uso do Gleam em casos de uso onde os recursos são restritos, como em IoT, ou quando se deseja produzir um arquivo executável estaticamente ligado (e autossuficiente), que não dependa da configuração presente no sistema para ser executado. Outros fatores associados aos ambientes de execução, como tempo de inicialização e tamanho dos programas, também prejudicam a aplicabilidade do Gleam nesses contextos: o instalador para a versão 27.0.1 da linguagem Erlang possui mais de 130 MB[4], tornando-a inviável em dispositivos com armazenamento limitado. Por outro lado, a utilização de um motor JavaScript, como o V8 do Chromium, poderia aliviar essa questão, mas ainda requereria vários megabytes.

Atualmente, o compilador Gleam implementa apenas a geração de código Erlang e JavaScript. Contudo, gostaríamos de poder executar código Gleam em um ambiente mais leve, que consumisse menos recursos e que estivesse também amplamente disponível. Um ambiente que se enquadra nesses requisitos é o WebAssembly[5] (ou Wasm), o qual é um conjunto de instruções e linguagem de programação idealizados como um alvo de compilação para linguagens de programação de alto nível, de modo que elas possam ser utilizadas na Web. O Wasm pode ser usado em contêineres, na nuvem, na Internet das Coisas e até em *desktops*, via APIs (*Application Programming Interfaces*) como o

[3] Rust. Disponível em: <https://rust-lang.org>
[4] Instalador disponível para *download* em: <https://www.erlang.org/downloads>
[5] WebAssembly. Disponível em <https://webassembly.org>

WebAssembly System Interface[6] (ou WASI), que permitem o acesso pelos módulos aos recursos do ambiente de execução (como entrada e saída e temporizadores).

1.2 Objetivos

Considerando as qualidades da linguagem Gleam, a linguagem Wasm, a interface WASI e a existência de ambientes de execução com baixa pegada de recursos para essas tecnologias, o objetivo principal deste trabalho foi implementar um gerador de código WebAssembly para a linguagem Gleam, capaz de compilá-la para arquivos que possam ser executados por ambientes Wasm com suporte às interfaces WASI. Como objetivos específicos deste trabalho, têm-se:

1. Mapear os recursos da linguagem Gleam para Wasm, e definir modos de traduzir código da primeira linguagem para a segunda;
2. Desenvolver uma versão preliminar de um gerador de código Wasm para a linguagem Gleam, e integrá-lo no compilador Gleam já existente;
3. Elaborar um relato compreensivo e didático do processo realizado, na forma desta monografia, que informe futuros colaboradores do projeto ou outros interessados na área.

1.3 Contribuições

Como principal contribuição deste trabalho, apresentamos uma implementação inicial de um gerador de código Wasm para o compilador da linguagem de programação Gleam. A nossa implementação abrange um subconjunto mínimo adequado para o ensino de linguagens de programação, abarcando variáveis, funções, casamento de padrões, tipos definidos pelo usuário, cadeias de caracteres, tipos numéricos, saída básica e outros recursos relacionados. O gerador de código produzido processa projetos de apenas um arquivo Gleam e emite um módulo Wasm correspondente, que pode ser fornecido a um ambiente de execução Wasm para ser executado.

Embora o conjunto de recursos implementados atualmente seja restrito, futuras versões do gerador de código permitirão a compilação de projetos Gleam inteiros em um único módulo Wasm estático e autocontido, capaz de ser executado em qualquer plataforma com suporte ao Wasm. Com este trabalho, contribuímos à comunidade uma implementação inicial e de código aberto de um gerador de código Wasm para parte da linguagem Gleam, além de comentários e observações de interesse para futuros trabalhos nesta área.

[6] WASI. Disponível em: <https://wasi.dev/>

Nossa abordagem também abre portas para uma maneira de distribuir código Gleam em executáveis estaticamente ligados: os dados do módulo Wasm da aplicação poderiam ser concatenados ao arquivo executável de um ambiente Wasm, e quando este fosse carregado, descompactaria os dados do módulo e os executaria. A existência de ambientes pequenos — como o WebAssembly Micro Runtime (WAMR) (Bytecode Alliance, 2024b), que, quando compilado, pesa menos de duzentos kilobytes — possibilita, quando unido à estratégia de concatenação dos dados, a geração de executáveis pequenos, ampliando as possibilidades de uso do Gleam em contextos onde a economia de recursos é importante. A implementação dessa estratégia em trabalhos futuros torna-se viabilizada pelo presente trabalho.

1.4 Organização do texto

O restante deste texto está organizado da seguinte maneira. Na Seção 2, apresentamos a fundamentação teórica utilizada no desenvolvimento deste trabalho, que inclui os conceitos principais de compilação, geração de código e as estruturas básicas das linguagens Gleam e Wasm, além de uma descrição em alto nível da interface WASI. Na Seção 3, relatamos o processo de desenvolvimento do gerador de código Wasm, fazendo um tratamento compreensivo de todos os recursos principais da linguagem Gleam e detalhando as regras de mapeamento entre as duas linguagens. Apresentamos uma avaliação crítica do trabalho e destacamos as dificuldades encontradas e trabalhos relacionados na Seção 4. Finalmente, na Seção 5, sumarizamos nossas contribuições e apresentamos perspectivas para trabalhos futuros.

2 Fundamentação

Nesta seção apresentamos conceitos e definições sobre linguagens de programação, compiladores, WebAssembly e Gleam. Primeiro fazemos um panorama sobre linguagens de programação e compiladores e, em seguida, detalhamos mais as linguagens Gleam e WebAssembly.

2.1 Linguagens de programação e compiladores

Linguagens de programação são notações formais utilizadas para descrever computações para seres humanos e computadores (AHO et al., 2007). As linguagens de programação são definidas, dentre outras características, em termos de sua sintaxe e semântica (BARBOSA, 2024): a sintaxe especifica a forma e as estruturas textuais válidas na linguagem, e a semântica determina qual é o significado de cada programa no contexto das regras definidas pela linguagem.

A sintaxe das linguagens de programação é usualmente definida em termos do formalismo da gramática livre de contexto (AHO et al., 2007), ou simplesmente gramática. A gramática é definida como um conjunto de regras, chamadas produções, as quais definem conversões entre símbolos de uma sequência para outros símbolos. As produções podem ser usadas recursivamente para gerar programas válidos em uma linguagem de programação, e por este motivo as gramáticas são também designadas por formalismos geradores. Um programa é dito pertencer à linguagem definida pela gramática (e, portanto, ser um programa sintaticamente válido) se existe uma sequência de aplicações das produções da gramática que gera o programa. O conjunto de todos os programas que podem ser produzidos por uma gramática é chamado de linguagem da gramática.

As linguagens definidas pelas gramáticas livres de contexto, denominadas linguagens livres de contexto, pertencem a uma hierarquia mais ampla de linguagens, chamada hierarquia de Chomsky (CHOMSKY, 1956), que pode ser vista na Figura 1.

Nem todas as propriedades de uma linguagem podem ser codificadas em uma gramática livre de contexto. Por exemplo, garantir que variáveis sejam declaradas em um arquivo antes de serem usadas requer, pelo menos, uma gramática sensível ao contexto (AHO et al., 2007), e garantir que um programa esteja corretamente tipado, em algumas linguagens como C++ e Java, requer uma gramática irrestrita (VELDHUIZEN, 2003; GRIGORE, 2017). Na prática, ainda é muito útil descrever linguagens de programação com gramáticas livres de contexto, devido à abundância de teoria e ferramentas desenvolvidas em torno desse tipo de gramática; as demais propriedades podem ser verificadas separadamente da estrutura sintática. Um programa escrito em uma linguagem é dito válido se ele concorda com as regras sintáticas e semânticas da linguagem de programação.

Por sua vez, compiladores são programas que traduzem código de uma linguagem

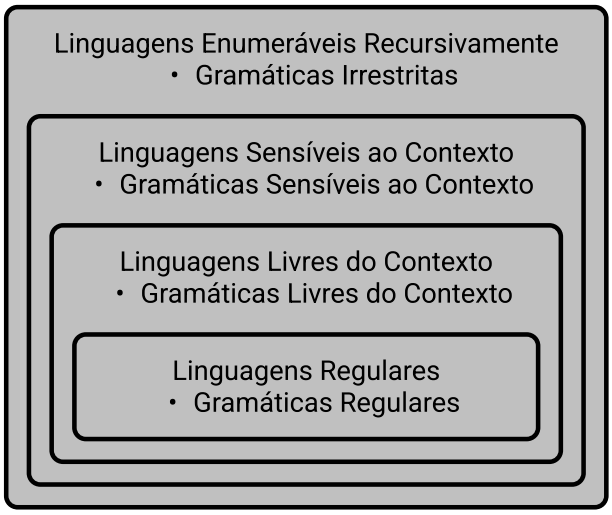

Figura 1 – Hierarquia de Chomsky com as gramáticas respectivas de cada nível.

de programação (fonte) para outra linguagem de programação (alvo) (AHO et al., 2007). Normalmente, a linguagem-fonte é uma linguagem de programação de alto nível, tal como C ou Pascal, e a linguagem-alvo é código de máquina, como x86-64 ou ARM, mas existem também compiladores que traduzem linguagens de programação de alto nível para outras linguagens de programação de alto nível, e compiladores que traduzem linguagens de montagem para outras linguagens de montagem.

O quão próximo uma linguagem de programação está do conjunto de instruções da arquitetura do computador determina se ela é uma linguagem de baixo nível, como linguagens de montagem, ou de alto nível, como C, Pascal e Rust. Código escrito em linguagens de alto nível costuma ser mais próximo do modo como seres humanos pensam e tratam de objetos mais próximos ao domínio de negócio; linguagens de baixo nível, em comparação, manipulam bem mais diretamente primitivas computacionais. Segundo Alan Perlis, um dos criadores do ALGOL, "uma linguagem de programação é de baixo nível quando os seus programas requerem atenção ao irrelevante" (PERLIS, 1982, tradução nossa). Os compiladores mais utilizados, como o GCC[7] e o Clang/LLVM[8], se ocupam mais com a tradução de linguagens de alto nível para linguagens de baixo nível do que traduções de baixo nível a baixo nível.

A implementação de compiladores é tradicionalmente dividida em duas fases que compreendem várias etapas: na vanguarda, ou *front-end*, o código-fonte da linguagem de programação passa pelas etapas de análise léxica, sintática e semântica e depois pela geração de representação intermediária; na retaguarda, ou *back-end*, a representação passa por várias etapas de otimização e, por fim, é usada na geração de código, que finalmente produz a linguagem-alvo (NYSTROM, 2021). A Figura 2 ilustra o fluxo tradicional de um

[7] GCC. Disponível em: <https://gcc.gnu.org/>
[8] Clang/LLVM. Disponível em: <https://clang.llvm.org/>

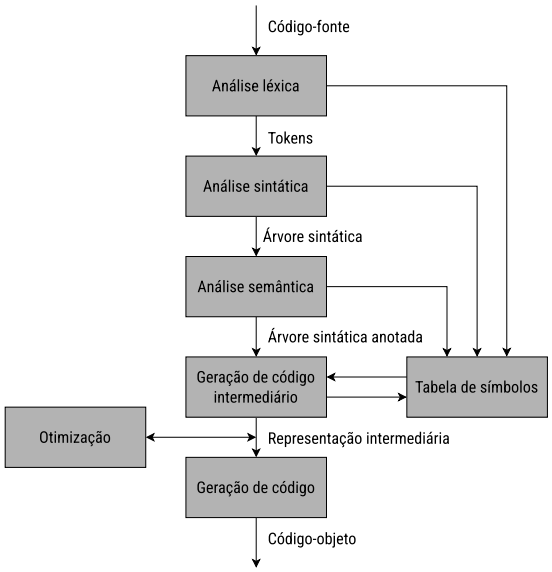

Figura 2 – Processo geral de compilação para uma linguagem de programação. Adaptado de SEBESTA (2011)

compilador (SEBESTA, 2011).

Na análise léxica, o código-fonte na linguagem-fonte é cortado e agrupado em símbolos de significado independente no texto, como números, identificadores e cadeias de caracteres. Uma sequência de símbolos (*tokens*) é gerada como saída. A análise léxica é geralmente definida em termos de uma gramática regular (Figura 1), a qual define os formatos e tipos de tokens que existem na linguagem.

Durante a etapa da análise sintática, os tokens são processados e organizados em uma árvore sintática, derivada da gramática, que é armazenada em memória. Algoritmos de análise sintática, tais como a descida recursiva e analisadores LR, podem ser utilizados na construção da árvore sintática.

Na análise semântica, a árvore sintática é visitada pelo analisador semântico e os tipos, coerções e vinculações são resolvidos e anotados na árvore. É nesta etapa que as propriedades da linguagem que não são codificadas na gramática são validadas.

As informações interpretadas durante as análises são armazenadas em uma estrutura de dados auxiliar chamada de tabela de símbolos, que associam informações como tipos, valores e outros detalhes aos elementos sintáticos do arquivo de entrada. Após as análises, o código original é convertido para uma representação intermediária, de um nível mais baixo que a linguagem original, mas de manipulação mais simples para o compilador. A representação é construída com base nas informações armazenadas na tabela de símbolos. Essa representação pode passar por otimizações, onde o código existente é trocado por código com semântica equivalente, mas com outras propriedades, como desempenho,

distintas.

Por fim, o gerador de código traduz o código na representação intermediária para código na linguagem-alvo. Para a tradução, é determinado o modo de uso dos registradores e a forma de acesso a estruturas em memória, entre outras coisas. Depois, o código-objeto é fornecido a um ligador, o qual produz um arquivo executável capaz de ser carregado pelo sistema operacional.

2.2 A linguagem Gleam

Gleam é uma linguagem de programação funcional projetada para escrita de sistemas concorrentes e escaláveis (PILFOLD, 2024a). Originalmente desenvolvida para a máquina virtual BEAM, da linguagem Erlang, atualmente também pode ser compilada para JavaScript.

A linguagem possui um sistema de tipos forte e estático do tipo Hindley-Milner (HINDLEY, 1969; MILNER, 1978). O compilador é capaz de inferir os tipos dos elementos no código sem que o(a) programador(a) os anote em cada declaração, deixando a linguagem com uma aparência de linguagem dinâmica, mas com a confiabilidade e expressividade de um sistema de tipos estático.

Gleam possui uma sintaxe parecida à de linguagens como Rust e C. Um exemplo de código Gleam pode ser visto na Listagem 1, que exibe um módulo que implementa uma calculadora de expressões aritméticas simples e demonstra boa parte dos recursos mais expressivos da linguagem, como tipos definidos pelo usuário, casamento de padrões, declarações de funções e inferência de tipos. A linguagem também permite a escrita de comentários, iniciando com duas barras (`//`).

```
1  import gleam/io
2
3  type Expr {
4    Add(Expr, Expr)
5    Sub(Expr, Expr)
6    Mul(Expr, Expr)
7    Div(Expr, Expr)
8    Const(Float)
9  }
10
11 fn eval(expr: Expr) -> Result(Float, String) {
12   case expr {
13     Add(lhs, rhs) -> op(lhs, rhs, fn (x, y) {x +. y})
14     Sub(lhs, rhs) -> op(lhs, rhs, fn (x, y) {x -. y})
15     Mul(lhs, rhs) -> op(lhs, rhs, fn (x, y) {x *. y})
16     Div(lhs, rhs) -> case eval(rhs) {
```

```
17        Ok(0.0) -> Error("Divisao por zero")
18        _ -> op(lhs, rhs, fn(x, y) {x /. y})
19      }
20      Const(f) -> Ok(f)
21    }
22  }
23
24  fn op(lhs: Expr, rhs: Expr, op: _) -> Result(Float, String) {
25    case eval(lhs), eval(rhs) {
26      Ok(lhs), Ok(rhs) -> Ok(op(lhs, rhs))
27      Error(e), _ -> Error(e)
28      _, Error(e) -> Error(e)
29    }
30  }
31
32  pub fn main() {
33    let expr = Add(Const(1.0),
34                   Mul(Const(3.0),
35                       Div(Const(4.0),
36                           Const(2.0))))
37    let assert Ok(result) = eval(expr)
38
39    // Efeitos colaterais apenas via bibliotecas.
40    io.debug(result) // 7.0
41  }
```

Listagem 1 – Exemplo de código Gleam

Componentes da linguagem

Aplicações Gleam são escritas com um conjunto de módulos, os quais são arquivos textuais contendo definições de funções, tipos e constantes. Além de definir seus próprios itens, um módulo também pode importar itens marcados como públicos (pela palavra reservada pub) de outros módulos (PILFOLD, 2024b). A Listagem 2 mostra um módulo que contém alguns exemplos de declarações e importações de itens de outros módulos.

```
1  /// Itens importados de outros módulos.
2  import gleam/io.{println} // função
3  import gleam/string // módulo
4  import gleam/option.{Some, type Option} // tipo e variante
5
6  /// Constantes.
7  const nome_opcional: Option(String) = Some("Lucas")
8  pub const pi = 3.141592
9
10 /// Tipos.
```

```
11 type Que {
12   Bar(x: Int)
13 }
14
15 /// Funções.
16 pub fn main() {
17   let assert Some(nome) = nome_opcional
18   io.debug(nome |> string.length |> Bar)
19   println(nome)
20 }
```

Listagem 2 – Itens de um módulo Gleam

Funções

As funções são definidas pelos seus parâmetros e pelo seu corpo, que é uma sequência de sentenças, e podem ser nomeadas ou anônimas. Quando declaradas no escopo de uma função, elas podem capturar variáveis locais a ela e se tornarem fechamentos (*closures*). As funções podem ser armazenadas em variáveis, passadas como parâmetros e retornadas de outras funções, o que as torna entidades de primeira classe (SEBESTA, 2011), assim como as funções nas demais linguagens de programação funcionais.

Tipos de dados

Os tipos pré-definidos em Gleam são: booleano (`Bool`), inteiro (`Int`), número de ponto flutuante (`Float`), nulo (`Nil`), cadeias de caracteres (`String`), listas (`List`), arranjos de bits (`BitArray`), um tipo resultado (`Result`) com duas variantes, `Ok` e `Error`, tuplas (`#(a_1, a_2, ..., a_n)`) e funções (`fn(arg_1, ..., arg_n) -> ret`).

Um tipo definido pelo usuário é uma união rotulada, ou tipo soma, de um conjunto de variantes, ou tipos produto. Cada variante pode conter uma sequência de campos, que podem ser acessados via casamento de padrões, ou acesso direto, caso aquele campo esteja presente na mesma posição, com o mesmo tipo e com o mesmo nome em todas as variantes.

A linguagem suporta polimorfismo paramétrico, permitindo a definição de funções e tipos parametrizados por outros tipos. Um exemplo de tipo que utiliza polimorfismo paramétrico é o `Result(ok, error)`, que pode ser visto na Listagem 1 como o tipo de retorno das funções `eval` e `op`. Os parâmetros dos tipos também são inferidos automaticamente pelo compilador, não sendo necessário o(a) programador(a) anotá-los em todas as declarações.

Controle de fluxo

A única estrutura de seleção definida pela linguagem é a expressão `case`, que verifica correspondência (casamento) entre valores e padrões. A expressão recebe um ou mais argumentos, ou sujeitos, a serem casados, e uma lista de casos, e executa o código do primeiro padrão que casa com os argumentos recebidos. Na Listagem 1 a função `op` (linhas 24–30) analisa dois valores do tipo `Result` para determinar se a avaliação da expressão foi bem-sucedida ou não, e, a depender das variantes dos tipos, selecionar o resultado. Na Listagem 3, exibimos outro exemplo de uma expressão case, que é feita sobre os restos da divisão de um número inteiro por três e por cinco.

A lista de casos da expressão deve ser exaustiva, isto é, deve cobrir todos os valores possíveis para os tipos dos argumentos passados, e o compilador Gleam verifica a exaustividade durante a análise semântica. Caso o(a) programador(a) não deseje fazer um tratamento de todas as possibilidades de maneira distinta, existe um padrão coringa, representado pelo caractere `_` (subtraço), que casa com qualquer valor incondicionalmente.

Uma cláusula de uma expressão case pode conter guardas, que condicionam a escolha do caso à avaliação da expressão de guarda como verdadeira. A linha 5 na Listagem 3 mostra uma expressão de guarda, e também demonstra o uso do padrão coringa.

```
1  import gleam/int
2
3  fn fale(x) {
4    case x % 3, x % 5 {
5      0, 0 if x % 7 == 0 -> "FIZZBUZZ!!"
6      0, 0 -> "FizzBuzz"
7      0, _ -> "Fizz"
8      _, 0 -> "Buzz"
9      _, _ -> int.to_string(x)
10   }
11 }
```

Listagem 3 – Casamento de padrões com inteiros

Como em algumas outras linguagens funcionais, Gleam não possui laços de iteração, então toda repetição deve ser expressa por recursividade. Funções recursivas são definidas, em geral, por um ou mais casos base e um ou mais casos recursivos. Em Gleam, a expressão `case` nos permite diferenciar entre estas situações. Um exemplo de uma função recursiva que computa a soma dos elementos de uma lista, utilizando casamento de padrões, é mostrado na Listagem 4.

```
1  import gleam/io
2  import gleam/int
```

```
3
4  fn soma(lst: List(Int)) -> Int {
5    case lst {
6      [] -> 0
7      [x, ..rest] -> x + soma(rest)
8    }
9  }
10
11 pub fn main() {
12   let res = soma([1,2,3]) // 6
13   io.debug("A soma é: " <> int.to_string(res))
14 }
```

Listagem 4 – Soma dos elementos de uma lista

O casamento de padrões também pode ser realizado na atribuição, como na linguagem Rust. Nos casos em que a atribuição não pode ser validada estaticamente, a linguagem disponibiliza a construção `let assert`, que adia a verificação do padrão para o tempo de execução, e gera um erro de execução caso não haja correspondência do padrão[9] A linha 37 da Listagem 1 contém um exemplo com o tipo `Result`. Detalhamos as atribuições mais adiante.

Além da expressão `case`, a linguagem implementa execução condicional via operadores `&&` (E) e `||` (OU): o operando à direita de um operador `&&` é apenas avaliado se o operando à esquerda é o valor lógico verdadeiro, e o operando à direita de um operador `||` é avaliado somente no caso em que o operando à esquerda é o valor lógico falso, isto é, os operadores são avaliados com curto-circuito.

Sentenças, expressões e atribuições

O corpo de uma função é um bloco. Um bloco é uma sequência de sentenças, que, por sua vez, podem ser expressões ou atribuições.

Expressões incluem operações unárias, binárias, constantes, acessos a variáveis, chamadas de funções, blocos, expressões *case* e outras estruturas. Cada expressão gera um valor quando avaliada; o valor de um bloco é definido como o valor gerado por sua última sentença, ou `Nil`, caso ele seja vazio.

As atribuições introduzem novas variáveis no ambiente de referenciamento em que estão inseridas. Gleam é uma linguagem com escopo léxico; blocos definem novos escopos, e a variável introduzida pela atribuição pertence ao escopo do bloco mais interno da

[9] Gleam busca minimizar o número de meios de gerar erros em tempo de execução, favorecendo o tratamento de erros por meio do seu tipo padrão `Result(ok, error)`. Um dos mecanismos é o `let assert`, mas outros incluem as expressões `panic` e `todo`, conforme detalhado na Seção 3.4.

declaração. Quando uma expressão referencia uma variável declarada em vários blocos, a ligação é feita com o bloco mais interno que contém uma declaração com aquele nome.

Uma vez declarada, uma variável não pode ter o seu valor alterado. A declaração de outra variável com o mesmo nome de uma anterior apenas oculta a original, preservando as vinculações a ela já existentes. Além disso, assim como as expressões, as atribuições também geram valores. O valor de uma atribuição é igual ao valor da expressão ao lado direito do seu símbolo de igual.

A gramática diferencia atribuições e expressões propositalmente. As atribuições podem aparecer apenas no interior de blocos, enquanto expressões podem aparecer em sentenças no geral. Esta restrição evita que construções como `func(3 + let x = 1)` sejam formadas, o que poderia confundir os usuários da linguagem a respeito do escopo no qual a atribuição `x` é válida.

Memória e execução

Os objetos em Gleam têm sua memória gerenciada automaticamente. A alocação de objetos e controle da sua memória dependem das implementações presentes nos ambientes de execução das linguagens-objeto.

A linguagem deixa vários detalhes de implementação em aberto para os ambientes de execução decidirem. Como um exemplo, além do gerenciamento de memória, o modo como os inteiros são representados é dependente da implementação: na máquina virtual BEAM, todos os números inteiros são *BigIntegers* (números inteiros de precisão arbitrária) por padrão, enquanto no JavaScript os números inteiros estão limitados aos que podem ser exatamente representados por um número de ponto flutuante de precisão dupla. Essa aparente falta de rigor dá mais margem de manobra para as implementações de outras retaguardas para a linguagem.

Um recurso da linguagem que a torna capaz de interagir com o mundo exterior são as funções e tipos com implementação externa, ou simplesmente funções e tipos externos. Normalmente, a linguagem Gleam não permite mudança de estado com as suas construções, mas nada impede que o ambiente externo mude o estado de variáveis externas. Tipos e funções declarados ao nível de um módulo podem ser marcados com uma anotação especial, `@external`, que sinaliza ao compilador a existência de uma implementação daquele tipo ou função em uma das linguagens alvo. Quando uma função externa é chamada, o compilador verifica se ela possui implementação externa, e caso sim, insere uma chamada a essa implementação ao invés da em Gleam. A anotação `@external` permite que o nome do módulo e o nome do item externo referenciado sejam especificados por meio do fornecimento de dois atributos. A Listagem 5 demonstra um exemplo de funções externas, que utiliza a função `console.log` do JavaScript.

```
// beep.gleam
@external(javascript, "./beep.mjs", "beep")
fn alert(x: String)

pub fn main() {
  alert("oi")
}

// beep.mjs
export function beep(x) {
    alert(x);
}
```

Listagem 5 – Exemplo de uso de funções externas

A implementação existente da linguagem Gleam contém um compilador capaz de emitir código para duas linguagens-alvo: Erlang e JavaScript. A ferramenta de linha de comando incluída com a implementação consegue compilar e executar o código gerado em um ambiente de execução da linguagem-alvo especificada, mas existe demanda da comunidade por um gerador de código alternativo, tal como para *WebAssembly*.

2.3 WebAssembly

WebAssembly, ou Wasm, é um conjunto de instruções e linguagem de programação projetado para ter um tamanho compacto e ser executado com alto desempenho, segurança, eficiência, portabilidade e inspecionabilidade na plataforma Web (W3C, 2024c). O Wasm foi criada pelo *World Wide Web Consortium* (W3C) como uma linguagem-alvo para que outras linguagens de programação de alto nível pudessem ser usadas na Web, como uma alternativa à compilação para JavaScript. Várias linguagens de programação possuem compiladores que emitem Wasm, entre elas, estão Rust, C e C++.

A linguagem Wasm foi projetada para ser executada em um ambiente hospedado. Devido à sua simplicidade, várias implementações de ambientes de execução existem, tanto de código aberto (Wasmtime[10], WAMR[11] e outros) quanto proprietários.

O código Wasm é organizado em módulos, os quais são unidades atômicas de carregamento, interpretação e compilação (W3C, 2024f). Os módulos contêm definições de funções, tipos, variáveis globais ao módulo, memórias — vetores não tipados de bytes — e tabelas de referências a funções, entre outros objetos. A linguagem é estruturada, e o código Wasm é organizado em funções e blocos.

[10] Wasmtime. Disponível em: <https://github.com/bytecodealliance/wasmtime>
[11] WAMR. Disponível em: <https://github.com/bytecodealliance/wasm-micro-runtime>

```
(module
  (func (export "multiply") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.mul))
```

Listagem 6 – Exemplo de código Wasm no formato textual

A especificação WebAssembly define dois formatos para módulos Wasm: o formato binário e o formato textual — *WebAssembly Text Format (WAT)*. O formato binário é uma codificação linear e compacta de um módulo (W3C, 2024e), enquanto o formato textual utiliza uma representação em expressões-S (W3C, 2024h), facilitando a leitura e escrita pelos programadores. Ambos os formatos são equivalentes, permitindo conversão livre entre eles sem perda de informações. Ferramentas oficiais estão disponíveis para realizar essa conversão de forma prática.

O formato textual permite a inserção de comentários, tanto de linha, iniciados por um ponto e vírgula (`;`), quanto de bloco, delimitados pelos caracteres `(;` e `;)`.

As instruções da linguagem têm seu funcionamento definido em termos de uma máquina virtual baseada em pilha. A Listagem 6 mostra a definição de um módulo Wasm no formato textual que exporta uma função `multiply`, a qual recebe dois inteiros de 32 bits como parâmetros e retorna o produto de ambos. A instrução `local.get` carrega uma variável local na pilha e a instrução `i32.mul` retira dois elementos do topo da pilha e empilha o resultado da multiplicação dos dois elementos que foram desempilhados.

O uso da pilha é verificado em tempo de compilação de maneira que a quantidade e os tipos dos elementos deixados na pilha devem ser consistentes com as declarações presentes no módulo — por exemplo, se uma função é declarada com `(result i32)`, ao fim da sua execução, a função deve deixar exatamente um inteiro de 32 bits no topo da pilha, independentemente do fluxo de execução tomado pelo seu código.

Na Listagem 6, também é possível perceber que as interfaces de um módulo são tipadas. Isto é uma característica importante que permite a verificação estática dos módulos e confere segurança à linguagem. O WebAssembly foi projetado para ser executado na Web, e os seus módulos são executados de maneira isolada, com toda chamada de função módulo–módulo ou módulo–ambiente verificada ao nível dos tipos. Além disso, todas as operações de indexação são verificadas, e operações inválidas geram falhas que abortam a execução do módulo (W3C, 2024d). O ambiente no qual o código é executado é controlado, e apenas os módulos e acessos permitidos pelo ambiente são disponibilizados e executados.

Existem vários meios de realizar controle de fluxo em Wasm. De forma mais geral, é possível declarar blocos de instruções aninhados e realizar saltos condicionais ou incondicionais, para fora desses blocos. Também é possível realizar um salto para o início

```
(module
  (type $typ.fac (func (param i64) (result i64)))
  (func $fun.fac (export "fac") (type $typ.fac)
    local.get $n
    i64.const 1
    i64.lt_s
    if (result i64)
      i64.const 1
    else
      local.get $n
      i64.const 1
      i64.sub
      call $fun.fac
      local.get $n
      i64.mul
    end))
```

Listagem 7 – Fatorial de um número em WAT

de um bloco através da instrução `loop`, que define um bloco e cria um rótulo no início desse, permitindo assim a construção de laços de repetição. Os blocos devem ser anotados dos efeitos que têm sobre a pilha de execução. Como conveniência, a linguagem define a instrução `if`, que implicitamente declara blocos e usa o valor no topo da pilha para decidir qual dos ramos será executado, de maneira semelhante às estruturas condicionais de outras linguagens de programação. Finalmente, Wasm permite realizar chamadas a funções, a partir das quais o fluxo de execução é transferido para a nova chamada. Existem instruções projetadas especificamente para realizar chamadas de funções em posição de cauda sem consumir memória adicional da pilha, mas não as aproveitamos em nosso trabalho por questões de tempo. A Listagem 7 mostra um exemplo de uso da instrução `if` no cálculo do fatorial de um inteiro.

O Wasm é desenvolvido por meio de propostas de extensão, que definem novos recursos para a linguagem. Um exemplo de extensão proposta à linguagem, que é utilizada neste trabalho, é o *Garbage Collection for WebAssembly* (W3C, 2024a), ou Wasm GC. O Wasm GC define uma série de novos tipos que têm a sua memória gerenciada automaticamente por um coletor de lixo. A extensão também possibilita que tipos estruturados sejam declarados e instanciados nos módulos, inclusive como subtipos de outros tipos, permitindo que sejam realizadas coerções e outras conversões. A proposta de extensão teve o seu produto mínimo viável (*MVP*) publicado no ano de dois mil e vinte e quatro, e o Wasm GC já é suportado nativamente nos quatro principais navegadores Web (Google Chrome, Apple Safari, Microsoft Edge e Mozilla Firefox).

Dois tipos de interesse particular da proposta Wasm GC são os tipos `struct` e `array`. O `struct` permite que uma sequência de tamanho fixo de tipos Wasm seja consolidada em um único objeto, enquanto o tipo `array` permite que um arranjo de valores de um mesmo tipo seja alocado linearmente e manipulado como um único objeto em

```
(module
  (type $String
    (array (mut i8)))
  (type $typ@eq@String
    (func (param (ref $String) (ref $String)) (result i32)))
  (type $typ@strsub@String
    (func (param (ref $String) i32 i32) (result (ref $String))))
  (type $sum@Teste (sub
    (struct (field i32))))
  (type $fun@int_to_string
    (func (param i32) (result (ref $String))))
  (type $typ@Teste.A (sub final $sum@Teste
    (struct (field i32) (field i32))))
  (type $typ@Teste.B (sub final $sum@Teste
    (struct (field i32))))
  ...)
```

Listagem 8 – Algumas definições de tipos GC

memória. A Listagem 8 mostra alguns exemplos de declarações de tipos do Wasm GC, extraídos de um módulo compilado pelo gerador de código desenvolvido nesse trabalho.

A proposta do Wasm GC foi concebida para permitir que linguagens que utilizam coletores de lixo não precisem trazer suas próprias implementações de um coletor de lixo quando compiladas para Wasm. Esta implementação passaria a ser de responsabilidade do ambiente. As características da extensão tornam o Wasm naturalmente adequado a linguagens como o Gleam, que possuem gerenciamento automático de memória.

Vários ambientes de execução de código aberto existem, como o V8 e o SpiderMonkey (os interpretadores de código dos navegadores Chromium e Mozilla Firefox, respectivamente), e o Wasmtime e o WAMR, desenvolvidos pela Bytecode Alliance (um consórcio de empresas de software interessado no desenvolvimento e promoção do WebAssembly como uma plataforma de aplicações).

Como consequência do seu projeto orientado à plataforma Web, o Wasm, por padrão, não define primitivas de entrada/saída e comunicação, salvo via funções externas declaradas e disponibilizadas previamente pelo ambiente. O ambiente pode disponibilizar funções de acesso a arquivos, à entrada e saída padrões e à rede de maneira controlada, tornando o Wasm útil em outros domínios que não o da Web, como em aplicações servidoras ou na execução de código não confiável (W3C, 2024j). Existem padrões, como o WASI, que definem interfaces comuns de execução.

2.4 WASI

Devido aos requisitos de segurança da Web, módulos Wasm são executados pelos ambientes de maneira isolada do mundo exterior. O *WebAssembly System Interface*, ou WASI, é uma coleção de interfaces de programação de aplicações (APIs) seguras (W3C,

2024i) que oferece comunicação e acesso a recursos do ambiente de execução a módulos Wasm, de maneira controlada, sem a necessidade de um intermediador Web/JavaScript. Um ambiente de execução Wasm que implementa o WASI permite que programas Wasm tenham capacidades semelhantes às de programas nativos.

Os padrões WASI definem uma série de tipos e funções que podem ser acessados por módulos Wasm. Alguns exemplos de APIs oferecidas pelo WASI são interfaces de fluxos de bytes, *sockets* TCP e UDP, acesso ao sistema de arquivos, geração de números aleatórios e realização de requisições e respostas HTTP (W3C, 2024b). Os padrões WASI em si especificam apenas as interfaces; os ambientes de execução Wasm/WASI são os responsáveis por implementá-los.

A especificação WASI continua em desenvolvimento e a sua última versão lançada é a WASI 0.2, de janeiro de 2024 (também chamada de *WASI Preview 2*). Alguns ambientes de execução de código aberto que suportam a interface são o Node.js[12], o Deno[13] e o WAMR. A versão 0.2 do WASI é definida em termos da proposta do modelo de componentes Wasm (Bytecode Alliance, 2024a), que especifica uma linguagem de descrição e composição de interfaces que permite que vários módulos Wasm possam ser integrados e se comunicarem entre si, de maneira encapsulada e segura.

[12] Node.js. Disponível em: <https://nodejs.org>
[13] Deno. Disponível em: <https://deno.com>

3 Desenvolvimento

Nessa seção descrevemos a implementação do gerador de código Wasm para a linguagem Gleam que desenvolvemos e incorporamos ao compilador Gleam. O desenvolvimento de um gerador de código pode parecer uma tarefa grande e intimidadora à primeira vista, mas ela, assim como outros problemas complexos em computação, pode ser dividida em subproblemas menores mais simples. Um gerador de código é um algoritmo que recebe como entradas um ou mais módulos na linguagem-origem, e produz um ou mais módulos na linguagem-destino, com o seu processamento realizando o mapeamento entre esses objetos. Por sua vez, estes objetos possuem componentes, que possuem sub-componentes e assim sucessivamente. Eventualmente, no nível mais baixo, a regra de tradução torna-se mais evidente.

Um modo como isso pode ser implementado é através de um conjunto de funções recursivas, responsáveis por traduzir elementos individuais do módulo Wasm. Por exemplo, haveria uma função que emite um literal, outra que traduz uma operação binária, uma que traduz uma expressão, mais outra que traduz uma função, e assim sucessivamente. No caso do Wasm, cuja semântica é orientada em torno de uma pilha de execução, a tradução do elemento sintático mais fundamental do Gleam, as expressões, é realizada de maneira direta por um passeio em pós-ordem nos nós da árvore sintática da expressão. Uma explicação mais aprofundada pode ser vista na Seção 3.4.

Os maiores desafios da implementação se resumem, então, a encontrar e definir regras de tradução devem ser usadas para cada tipo de item, e como esses elementos traduzidos devem ser compostos no módulo resultante com as demais informações do compilador. As regras de tradução adotadas são explicadas na Seção 3.4, e o modo como o contexto e as informações são gerenciados é definido na Seção 3.3.

Para inserirmos o gerador de código no compilador, precisamos adaptar o código existente para que ele invocasse as novas funções. O modo como isto foi realizado é detalhado na Seção 3.2. Por fim, a Seção 3.5 relata como os itens traduzidos são agrupados e serializados em um módulo Wasm.

Neste trabalho, realizamos estas alterações de modo incremental, conforme ilustra a Figura 3. Primeiro, identificamos as mudanças necessárias no compilador e as implementamos. Depois, implementamos porções pequenas da linguagem por vez, partindo de uma implementação mínima que apenas compilava expressões matemáticas, até chegar ao conjunto de recursos apresentados neste trabalho. Por fim, a cada ciclo, testamos os recursos introduzidos e corrigimos erros encontrados.

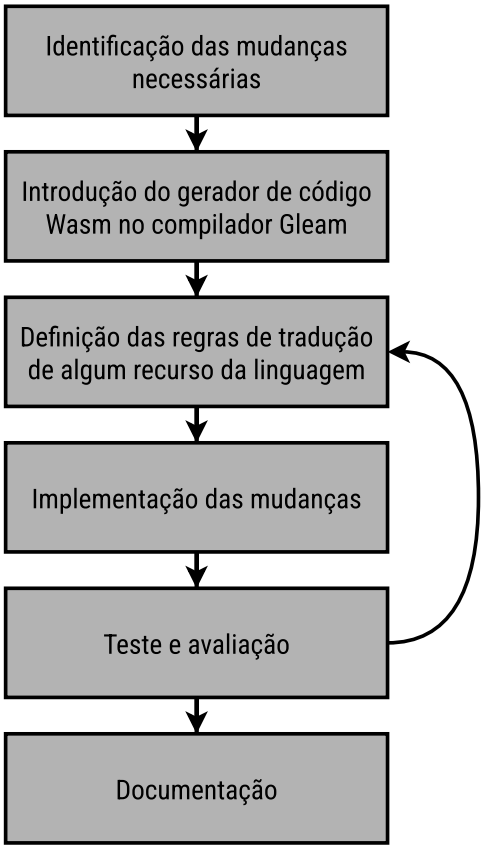

Figura 3 – Metodologia de implementação do gerador de código.

3.1 Ferramentas

Para desenvolver o trabalho, aproveitamos o rico ecossistema de ferramentas das linguagens Wasm e Rust como suporte. As ferramentas disponíveis nos auxiliaram na construção e execução de módulos Wasm e na conversão dos módulos para um formato textual, adequado para leitura e análise manual. Descrevemos as ferramentas e linguagens de programação que utilizamos a seguir.

A implementação do emissor de código foi realizada na linguagem de programação Rust, que é a linguagem de implementação original do compilador Gleam. O ambiente de desenvolvimento utilizado foi o editor de texto Visual Studio Code[14], com a extensão rust-analyzer.

Utilizamos as ferramentas *wasm-tools*[15] para inspecionar o código gerado pelo nosso gerador. O *wasm-tools* possui capacidade de ler módulos no formato binário e no formato textual do Wasm, realizar a conversão entre os dois e validá-los. Além disso, ele possui utilitários que imprimem informações sobre as seções e itens de um módulo.

[14] Visual Studio Code. Disponível em: <https://code.visualstudio.com/>
[15] wasm-tools. Disponível em: <https://github.com/bytecodealliance/wasm-tools>

Para a geração do código Wasm, utilizamos a biblioteca de código aberto *wasm-encoder*, disponível no repositório de pacotes Rust *crates.io*. A biblioteca fornece uma interface tipada em Rust para a escrita e composição de módulos Wasm no formato binário. As funções do *wasm-encoder* operam em um baixo nível, escrevendo os bytes de cada seção diretamente.

O código emitido utiliza as interfaces do padrão WebAssembly System Interface para comunicação com o sistema operacional, como em nossas funções de saída, descritas posteriormente nesta seção.

Para executar o código Wasm gerado, utilizamos o ambiente de execução Wasmtime, devido ao seu suporte às interfaces WASI e ao Wasm GC e por ser uma implementação de referência para a linguagem.

3.2 O compilador Gleam

O compilador Gleam é uma aplicação escrita na linguagem de programação Rust, e sua função é compilar projetos Gleam para uma linguagem-alvo selecionada, que na versão oficial pode ser JavaScript ou Erlang. Um projeto Gleam contém pacotes, os quais são conjuntos de módulos associados a metadados que indicam seus detalhes e dependências. Durante a compilação de um projeto, o compilador realiza os seguintes passos:

1. Analisa e armazena as opções de compilação passadas pelo usuário, tais como a linguagem-alvo desejada, em uma estrutura de dados;
2. Lê o arquivo de metadados do projeto e carrega os seus dados em memória;
3. Realiza o *download* de cada pacote especificado como dependência nos metadados do projeto, a partir do repositório de pacotes do ecossistema Erlang, Hex;
4. Compila os pacotes baixados e o pacote principal do projeto e salva o resultado no sistema de arquivos.

Por sua vez, as etapas de compilação de um pacote, realizadas pelo compilador, são as seguintes:

1. Lê os arquivos do pacote e os armazena em memória;
2. Instancia um módulo para cada arquivo lido;
3. Realiza as etapas de análise léxica e sintática dos módulos e sintetiza uma árvore sintática abstrata (*Abstract Syntax Tree*, AST) não tipada, contendo lacunas a serem preenchidas durante a análise semântica;

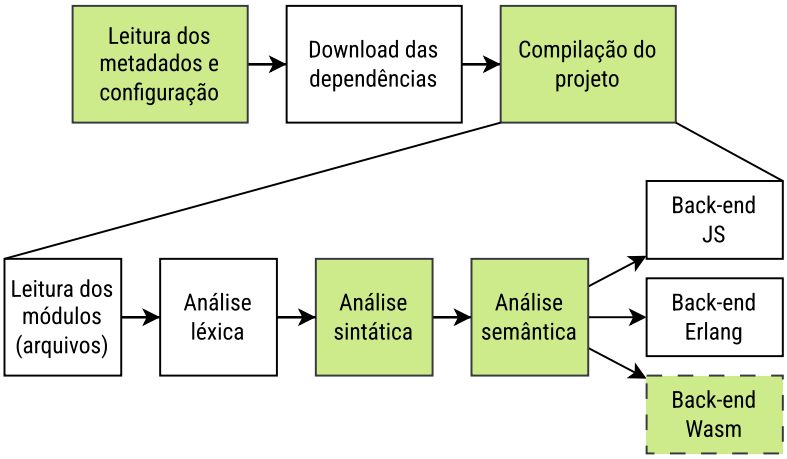

Figura 4 – Estrutura em alto-nível do compilador Gleam.

4. Executa a análise semântica sobre a AST não-tipada, e os tipos do código são inferidos, verificados e anotados na árvore sintática. Algumas estruturas de açúcar sintático são convertidas para as suas formas canônicas. O analisador também verifica as funções externas no código-fonte, determinando se o código delas pode ser executado com a linguagem-alvo selecionada pelo usuário. Ao fim, gera uma AST tipada;

5. Finalmente, o compilador passa a AST tipada para o gerador de código selecionado pelo usuário. O resultado é escrito no sistema de arquivos virtual.

Estas etapas estão ilustradas na Figura 4, em forma levemente simplificada. Na figura, ainda destacamos os pontos em que realizamos alterações no compilador para acomodar o novo gerador em verde. Para o processo de compilação de projetos, alteramos o passo 1, para incluir a linguagem Wasm como uma das opções. Já para o processo de compilação de pacotes, modificamos as etapas 4 e 5. Na etapa 4, adicionamos suporte a implementações externas Wasm, e o nosso gerador foi adicionado como uma alternativa à etapa 5, acionada a depender da opção escolhida pelo usuário no momento da compilação do projeto.

3.3 A estrutura de um gerador de código

Quando um gerador de código é acionado, ele recebe apenas as árvores sintáticas tipadas dos módulos sendo compilados e uma estrutura de acesso ao sistema de arquivos, sem nenhuma outra informação adicional. Os outros dois geradores processam esses módulos e salvam as suas versões traduzidas para os seus alvos no sistema de arquivos. Nosso gerador funciona da mesma forma: ele recebe a árvore sintática tipada de um módulo Gleam e produz arquivos contendo módulos Wasm na saída.

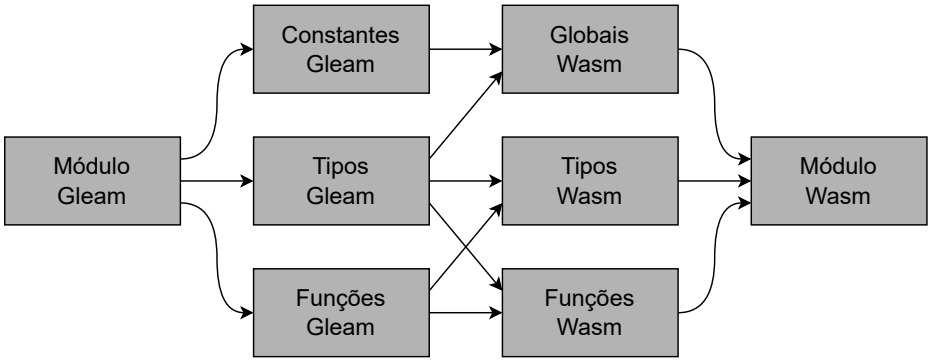

Figura 5 – Mapeamento dos itens de um módulo Gleam para um módulo Wasm.

Uma forma para implementar o mapeamento de um módulo Gleam para um módulo Wasm surge a partir de uma análise mais cuidadosa dos conteúdos de cada módulo. Módulos Gleam são, em essência, um conjunto de definições de constantes, funções, tipos e de itens importados. Módulos Wasm, por sua vez, declaram tipos, funções e globais, dentre outros itens. O fato dos módulos possuírem conceitos com os mesmos nomes não é coincidência; há bastante sobreposição entre o significado destes elementos.

Uma abordagem natural, portanto, é mapear os conceitos de uma linguagem aos seus correspondentes na outra: constantes Gleam para globais Wasm, funções para funções e tipos para tipos. Existem ressalvas: funções Gleam possuem um tipo também, e os tipos definidos pelo usuário podem gerar instâncias globais, construtores e comparadores de igualdade. Adotamos esse mapeamento em nossa implementação na expectativa de que, ao reaproveitar semelhanças entre as duas linguagens o máximo possível, as regras de tradução se tornariam mais simples. A Figura 5 mostra o modo como o mapeamento foi realizado.

Em resumo, nosso gerador implementa um processo que toma um módulo Gleam, analisa suas definições, emite definições correspondentes em Wasm e as junta em um módulo Wasm final. Durante esse processo, queremos que a tradução seja semanticamente equivalente, o que significa que o código gerado deve ter os mesmos efeitos e significado que o código original. Isso implica que devemos manter os modelos de execução das duas linguagens em mente conforme estabelecemos as regras de tradução.

Contudo, como mencionado no início desta seção, a geração de código é sensível ao contexto em que está inserido: aos nomes em escopo, às definições do usuário, aos tipos das variáveis referenciadas, e assim por diante. O gerador desenvolvido evita modificar o código após gerado, o que significa que no momento em que este está sendo escrito, todas as informações de tipo e escopo precisam estar disponíveis. Por exemplo, cada função Wasm possui um identificador numérico único, ou índice, usado no momento da sua chamada. Isso significa que, minimamente, o índice Wasm correspondente a uma função Gleam precisa ser conhecido no instante em que esta é chamada.

Para tal fim, mantemos uma estrutura de dados com informações sobre as entidades sendo referenciadas no código. Esta estrutura é a tabela de símbolos, e nela armazenamos informações sobre tipos Wasm, funções, tipos definidos pelos usuários e suas variantes, constantes e variáveis locais. A Listagem 9 mostra a estrutura da tabela e a estrutura de uma de suas entradas, as funções.

```
pub struct Type {
    pub id: TypeId,
    pub name: EcoString,
    /// A definição Wasm do tipo em si.
    pub definition: TypeDefinition,
}

pub struct Function {
    pub id: FunctionId,
    pub name: EcoString,
    pub signature: TypeId,
    pub arity: u32,
}

// ...

pub struct SymbolTable {
    pub types: Store<Type>,
    pub functions: Store<Function>,
    pub sums: Store<Sum>,
    pub products: Store<Product>,
    pub constants: Store<Constant>,

    // ...
}
```

Listagem 9 – Definição da tabela de símbolos e de informações de função.

O modo como a implementamos é o seguinte: para cada tipo de entidade no código, definimos um vetor de estruturas com informações sobre cada instância da entidade. O vetor é acessado com um índice identificador único, incrementado automaticamente toda vez que uma nova entidade é gerada. Quando uma entidade referencia outra, armazenamos apenas o índice da entidade referenciada, e quando necessário, usamos o índice para acessar a entidade correspondente na tabela de símbolos. O registro de itens na tabela é feito em dois passos: primeiro, um índice novo para o elemento deve ser gerado, e segundo, os dados devem ser inseridos na tabela com aquele índice.

Considerando estas restrições, o modo como o código é gerado segue os seguintes passos, considerando as regras de mapeamento entre as duas linguagens:

1. O módulo é varrido e, para cada item em seu nível superior (tipos, funções e constantes), um índice é gerado e registrado na tabela de símbolos.

2. Os tipos de cada elemento do módulo são gerados e registrados, referenciando outros tipos se necessário através dos seus índices.

3. As instruções Wasm são geradas para as funções do módulo Gleam, usando os índices e tipos armazenados nos passos anteriores.

Esta abordagem tem algumas limitações, que destacamos na Seção 4. Descrevemos como cada parte da linguagem Gleam foi traduzida nas seções seguintes.

3.4 Definição das regras de tradução

Tipos de dados

Os tipos predefinidos pela linguagem Gleam são `BitArray`, `Bool`, `Float`, `Int`, `List`, `Nil`, `Result`, `String` e `UtfCodepoint`, que representam vetores de bits, valores booleanos, números de ponto flutuante, números inteiros, listas encadeadas, um tipo nulo, um tipo resultado, uma cadeia de caracteres e um ponto de código *Unicode*, respectivamente. Destes tipos, implementamos apenas `Bool`, `Float`, `Int`, `Nil` e `String`.

Escolhemos, para o `Int`, o tipo `i32` da especificação Wasm, o qual é um número inteiro de 32 bits. O Wasm não diferencia entre tipos com sinal e sem sinal, sendo a única diferença entre eles as instruções usadas para realizar as operações. Optamos por esta representação porque o Gleam não especifica restrições de precisão mínima e máximas para inteiros — no Erlang eles possuem precisão arbitrária, e no JavaScript eles são implementados como números de ponto flutuante de precisão dupla — e porque o modelo WASI32 utiliza inteiros de 32 bits, então evitamos conversões desnecessárias.

Para o tipo `Float`, adotamos o número de ponto flutuante de precisão dupla `f64` do Wasm, que segue o padrão IEEE 754 (IEEE..., 2019). Esta é a mesma representação utilizada nos geradores de Erlang e JavaScript. Quando encontramos um literal inteiro ou um número de ponto flutuante, emitimos as instruções `i32.const` e `f64.const`, respectivamente.

O tipo `String` é mapeado para um arranjo de bytes, e a representação adotada é o padrão UTF-8 (YERGEAU, 2003). O UTF-8 é um padrão que mapeia pontos de código Unicode para sequências de um a quatro bytes. O modo como representamos o arranjo é através do tipo do Wasm GC `array`. Para poupar espaço no arranjo, armazenamos os bytes das cadeias de caracteres de maneira densa, com um recurso chamado de *packed storage*. Um arranjo no Wasm GC consegue armazenar objetos com trinta e dois bits de comprimento ou mais, mas o armazenamento denso empacota múltiplos inteiros de

comprimento menor em um único inteiro maior. O modo de acesso desses elementos é feito com instruções que os desempacotam: `array.get_u`, que expande o inteiro assumindo que ele não possui sinal, e `array.get_s`, que expande o inteiro assumindo que ele possui sinal.

O Wasm GC disponibiliza alguns meios de inicializar arranjos. O meio que utilizamos para inicializar strings é armazenando os literais em itens `data` no módulo Wasm, e inicializando os arranjos a partir deles. Um item `data` contém uma sequência de bytes e é normalmente utilizado em outros compiladores Wasm para inicializar memória estática e outras estruturas de dados pré-populadas. A Listagem 10 mostra um exemplo básico de um módulo Wasm que carrega um literal `String` "Olá, mundo!" em um arranjo. Construímos o tipo `array` utilizando a instrução `array.new_data`, que recebe como parâmetro o índice de um item `data` e retira da pilha o comprimento dos dados e um deslocamento em relação ao início do item `data` selecionado. A instrução copia os dados especificados para o arranjo criado.

```
1  (module
2    ;; os números nos comentários abaixo representam os índices
3    ;; das entidades no módulo, e são inseridos automaticamente
4    ;; pelo conversor do wasm-tools
5    (type $String (;0;) (array (mut i8)))
6    (type $fun@main (;1;) (func (result (ref $String))))
7    (func $main (;0;) (type $fun@main) (result (ref $String))
8      ;; Endereço em data
9      i32.const 0
10     ;; Quantidade de bytes
11     i32.const 12
12     ;; Criação da instancia
13     array.new_data $String 0
14   )
15   (data (;0;) "Ol\c3\a1, mundo!")
16 )
```

Listagem 10 – Exemplo de um módulo com um literal do tipo String

Gleam permite que os seus usuários definam novos tipos, que são uniões rotuladas de variantes. Em Wasm, representamos isso da seguinte maneira: para cada tipo definido pelo usuário (TDU) Gleam, definimos um `struct` "pai" e um `struct` para cada variante do tipo. Os `structs` das variantes possuem como campos um discriminante (inteiro de trinta e dois bits que indica a variante) e os campos declarados no código Gleam, e o `struct` pai contém o discriminante e todos os campos com mesmo nome e posição em todas as variantes. A passagem de diferentes variantes no código Wasm é possível graças às regras de subtipagem definidas na especificação do GC, que estabelecem que um tipo `struct` é considerado subtipo de outro `struct` se os campos deste último constituírem

um prefixo dos campos do primeiro, independentemente se a relação de subtipagem foi declarada de modo explícito ou não.

Esta regra demonstra que o sistema de tipos do Wasm é estrutural, diferentemente do sistema de tipos do Gleam, que é nominal. Em um sistema nominal, dois tipos são considerados iguais se possuem o mesmo nome, enquanto num sistema estrutural, dois tipos são considerados iguais se eles possuem a mesma estrutura interna. Isso justifica a adoção do campo discriminante nas representações dos TDUs, pois caso contrário, se um programa Gleam contivesse um tipo definido pelo usuário com duas variantes contendo os mesmos campos, seria impossível diferenciá-los em tempo de execução, visto que para o sistema de tipos do Wasm eles pertenceriam ao mesmo tipo.

Para determinar a variante de uma instância de um TDU, verificamos a qual das variantes o valor do discriminante corresponde. Não precisamos verificar se o tipo Gleam em si é o mesmo, pois a etapa de análise semântica já realiza esta verificação. Caso a variante corresponda, realizamos uma conversão do tipo da referência para o subtipo desejado, e então acessamos os seus campos normalmente.

Para cada variante, também definimos uma função Wasm construtora para ela, que tem como parâmetros os mesmos campos contidos na definição e retorna uma referência para uma instância do tipo da variante sendo construída. Nos casos em que a variante não possui campos, além do construtor, também declaramos uma variável global contendo uma instância do tipo. A variável global é populada na função de inicialização do módulo Wasm (`start`), a qual é executada antes da função principal começar a rodar. Todos esses tipos e funções são armazenadas na tabela de símbolos.

Para dar suporte à sintaxe de acesso direto de membro comum, explicada mais adiante nesta seção, nós reordenamos os campos dos TDUs de modo que todos os campos comuns apareçam no início das estruturas. O Gleam considera um membro que ocorre em todas as variantes com o mesmo tipo, nome e posição como membro comum, independentemente se o campo está no início, meio ou fim das variantes, mas o Wasm exige que os campos comuns estejam no início, tornando a reordenação necessária. Para dar suporte a esse recurso, mantemos na tabela de símbolos um mapeamento das posições originais dos membros para as posições reordenadas.

A Listagem 11 mostra a tradução de um TDU em uma sequência de tipos Wasm.

```
1 type Expr {
2     Add(l: Float, r: Float)
3     Neg(l: Float)
4     Const(c: Float)
5 }
```

Código contendo uma declaração de um TDU

```
(module
  (; ... ;)
  (type $sum@Expr
    (sub
      (struct
        ;; Identificador da variante
        (field i32))))
  (type $typ@Expr.Add
    (sub final $sum@Expr
      (struct
        ;; Identificador da variante
        (field i32)
        ;; Campo l
        (field f64)
        ;; Campo r
        (field f64))))
  (type $typ@Expr.Neg
    (sub final $sum@Expr
      (struct
        ;; Identificador da variante
        (field i32)
        ;; Campo l
        (field f64))))
  (type $typ@Expr.Const
    (sub final $sum@Expr
      (struct
        ;; Identificador da variante
        (field i32)
        ;; Campo c
        (field f64))))
  (; ... ;)
)
```

Listagem 11 – Código Wasm contendo tradução do TDU declarado

Para os tipos pré-definidos `Bool` e `Nil` optamos por por uma representação com inteiros de 32 bits (`i32`), isto porque os valores booleanos são assim representados em Wasm e instruções condicionais esperam um `i32` contendo o valor lógico. Representamos o `True` como o inteiro 1 e o `False` e `Nil` como o inteiro 0.

Ainda não implementados os tipos `Result`, `List`, `BitArray` e `UtfCodepoint`, mas descrevemos algumas abordagens possíveis para estes tipos na Seção 4

A seguir discutimos como as diversas operações sobre valores, como operações aritméticas, acesso a registros, chamadas de funções, foram implementadas. Começamos com um descrição do funcionamento da máquina virtual Wasm.

A máquina virtual Wasm

O modelo de execução do Wasm é descrito em torno da sua máquina virtual, que é uma máquina abstrata baseada em pilha. A máquina virtual armazena todos os valores empilhados por instruções, rótulos de destino que podem ser alcançados por desvios, e os registros de ativação das funções sendo executadas. Como um detalhe de implementação, além da pilha, o Wasm também define uma tabela que contém dados de todos os objetos alocados que podem ser referenciados e manipulados por código Wasm, permitindo a verificação em tempo de execução dos tipos das referências, dos tamanhos de arranjos, entre outras informações. Outros detalhes da máquina virtual, como um ponteiro de instrução, não são acessíveis ao código Wasm (W3C, 2024g).

Quando uma instrução é executada, ela realiza operações sobre a pilha e sobre a tabela de objetos (também chamada de *store*). Tais operações incluem empilhar e desempilhar objetos, declarar novos objetos na tabela, mudar propriedades de objetos existentes e invocar recursivamente outras instruções.

Avaliação de expressões na pilha

A maioria de código Gleam é composta de expressões, as quais são estruturas sintáticas que geram algum valor quando avaliadas. As expressões podem conter como operandos outras expressões, conforme a gramática da linguagem especifica.

No modelo de execução em pilha, as operações são executadas desta maneira: é necessário empilhar primeiramente os operandos antes de executar a instrução. Sendo assim, avaliamos a árvore sintática da expressão em um passeio pós-ordem, e essa sequência é refletida no código gerado. Como as expressões são tipos recursivos, esta visita é implementada de maneira recursiva.

A Listagem 12 mostra a função `math`, que realiza uma sequência de operações aritméticas sobre literais numéricos, e o código Wasm correspondente. A listagem também mostra a evolução da pilha durante a execução, onde é possível observar que os valores das operações de menor precedência permanecem na pilha até que as expressões de maior precedência são resolvidas. Isto é uma característica deste modelo de execução.

```
1 fn math() {
2   1 + { 3 - 5 } * 2
3 }
```

Código contendo uma expressão matemática

```
(module
  (type $fun@math
    (func (result i32)))
  (func $math (type $fun@math) (result i32)
    i32.const 1
    i32.const 3
    i32.const 5
    i32.sub
    i32.const 2
    i32.mul
    i32.add)
)
```

```
1
1 3
1 3 5
1 -2
1 -2 2
1 -4
-4
```

Listagem 12 – Código Wasm que avalia a expressão matemática

Para compilar uma expressão, precisamos decidir regras de tradução para cada um dos seus casos. Expressões podem ser literais, operações, expressões de exceção, estruturas de acesso, estruturas sintáticas, e outros.

Os literais incluem números inteiros, números de ponto flutuante, strings, funções anônimas, listas, tuplas e `BitArrays`. Destes, implementamos os literais numéricos e strings. O modo como estes literais são traduzidos e empilhados na pila foi explicado anteriormente nesta seção.

As operações, por sua vez, se dividem em unárias, binárias e n-árias, e incluem a negação de números e valores lógicos; operações aritméticas como soma, subtração, multiplicação e divisão; operadores lógicos "e" (`&&`) e "ou" (`||`); operadores relacionais como maior, menor, maior ou igual, menor ou igual, igual e não-igual; chamadas de funções; acessos diretos a membros comuns de TDUs; acessos a membros de tuplas; e atualizações de registros (não implementamos essas duas últimas operações). Todas essas operações são traduzidas primeiro avaliando-se o resultado dos seus operandos e os deixando na pilha, e depois executando a operação.

Em especial, para as divisões, emitimos uma instrução `if` que verifica se o divisor é igual a zero, e nesse caso é empilhado um zero. Além disso, a igualdade e a desigualdade são definidas para os TDUs também; para dar suporte a esse recurso, para cada TDU, geramos funções que testam a igualdade campo-a-campo de duas instâncias. Caso algum dos campos seja um TDU, fazemos isso recursivamente. As instruções que implementam a comparação de igualdade entre dois valores são selecionadas no momento da geração da expressão de acordo com os tipos dos operandos sendo avaliados.

Também existem as expressões `todo` e `panic`, que geram uma falha quando executadas, opcionalmente com uma mensagem exibida na saída padrão (não implementamos este recurso). No Wasm, mapeamos essas expressões para a instrução `unreachable`, que

gera um erro e encerra a execução do módulo atual.

Armazenando e acesso a variáveis e tipos

Além de manipular literais, podemos acessar itens nomeados na linguagem Gleam. Estes itens incluem parâmetros de funções, variáveis locais, funções e tipos do módulo, entre outros. Para implementar este conceito, precisamos de alguma forma representar o contexto em que uma expressão está sendo compilada — esta estrutura é o ambiente de referenciamento.

Gleam é uma linguagem de escopo léxico. Nomes introduzidos em um bloco podem ser acessados em um escopo que começa na linha onde a variável foi declarada e que se estende até o final do bloco. Nomes podem ser introduzidos por declarações, as quais são outro tipo de sentença além da expressão: uma declaração atribui ao(s) nome(s) no lado esquerdo da igualdade o valor da expressão no seu lado direito, e uma vez atribuídos, os valores não podem mais ser alterados. Além das declarações, definições de constantes, tipos e funções ao nível do módulo também implicitamente introduzem esses nomes no escopo global.

Para lidar com essa característica, implementamos o ambiente de referenciamento como um mapa que associa nomes a índices de objetos, acompanhado de uma referência opcional a um ambiente de referenciamento envolvente. Ao iniciar o processamento de um bloco, criamos um novo ambiente de referenciamento que aponta para o ambiente ativo. Novas declarações introduzem nomes no ambiente atual. Quando um nome é referenciado, verificamos no mapa se há uma ligação dele a um objeto; caso contrário, a busca é encaminhada para o ambiente envolvente, e assim sucessivamente, até que o escopo global é alcançado ou uma ligação encontrada. Este mecanismo permite que declarações feitas em ambientes mais internos ocultem declarações de mesmo nome em ambientes mais externos. A Listagem 13 mostra estas características na declaração da estrutura que armazena as ligações em Rust, reproduzida aqui de maneira simplificada.

Observamos que o espaço de nomes em Gleam é dividido em dois: o espaço dos tipos e o espaço dos valores. Os dois espaços são sempre acessados em contextos sintaticamente distintos e, portanto, podem conter ligações diferentes com o mesmo nome. Isto ocorre, por exemplo, quando um TDU é declarado com uma única variante que possui o mesmo nome do tipo, algo permitido em Gleam.

Na Listagem 13, vemos que referências a locais devem estar acompanhadas de um identificador. Todas as variáveis locais precisam ser declaradas antecipadamente em uma função Wasm e devem ser acessadas pelos seus índices numéricos. Para gerar o código que suporta uma série de declarações, mantemos uma estrutura semelhante à tabela de símbolos, mas apenas para as variáveis locais da função sendo compilada. Quando uma nova variável local é solicitada, gravamos o seu tipo na estrutura e geramos um índice,

```rust
pub enum Binding {
    Local(LocalId),
    Function(FunctionId),
    Product(ProductId),
    Constant(ConstantId),
    Builtin(BuiltinType),
}

pub enum TypeBinding {
    Sum(SumId),
}

pub enum BuiltinType {
    Nil,
    Boolean { value: bool },
}

pub struct Environment<'a> {
    values: HashMap<EcoString, Binding>,
    types: HashMap<EcoString, TypeBinding>,
    enclosing: Option<&'a Environment<'a>>,
}
```

Listagem 13 – Declaração da estrutura do ambiente de referenciamento

que então é inserido no ambiente de referenciamento juntamente com o nome da variável.

A Listagem 14 mostra um exemplo de uma função que declara e manipula variáveis Gleam e o seu código Wasm respectivo, ligeiramente simplificado do formato emitido por nosso gerador de código.

```rust
fn vars() {
  let x = 1
  {
    x
    let x = 2
    x
  }
  x
}
```

Declaração e acesso a variáveis

```
(module
  (type $fun@vars
    (func (result i32)))
  (func $vars (type $fun@vars) (result i32)
    (local $x i32)
    (local $x2 i32)

    ;; (L2) let x = 1 (declara x externo)
    i32.const 1
    local.set $x
    local.get $x
    drop

    ;; (L3) { (abre novo escopo)

    ;; (L4) x (referencia o x externo)
    local.get $x
    drop

    ;; (L5) let x = 2 (declara o x interno)
    i32.const 2
    local.set $x2
    local.get $x2
    drop

    ;; (L6) x (referencia o x interno)
    local.get $x2
    drop

    ;; (L7) } (fecha bloco, x interno sai de escopo)

    ;; (L8) x (referencia o x externo)
    local.get $x
  )
)
```

Listagem 14 – Código Wasm com variáveis

Casamento de padrões e a expressão case

As expressões de estruturas sintáticas incluem os blocos, a expressão `case` e os `pipelines`. Dessas estruturas, apenas os `pipelines` não foram implementados.

A expressão `case` é a única estrutura condicional em Gleam, e um dos poucos mecanismos de controle de fluxo da linguagem (além dos operadores lógicos e das chamadas de funções). Uma expressão `case` é composta por uma lista de valores e uma sequência de cláusulas. As cláusulas possuem um ou mais padrões e uma expressão consequente,

avaliada caso o padrão case com o sujeito correspondente. O modo como ela é avaliada é pela realização do casamento dos valores fornecidos com os padrões especificados.

No casamento de padrões, a grosso modo, o objetivo é determinar se a estrutura de um objeto sob avaliação, chamado de sujeito, corresponde à estrutura de uma expressão simbólica qualquer, chamada de padrão. O problema pode ser visto como uma restrição do problema de unificação simbólica, onde um dos lados da equação é constante e não possui variáveis (o lado do sujeito) enquanto o outro pode possuir variáveis livres (o lado do padrão) (DOWEK, 2001). Assim, o desafio é encontrar valores para as variáveis livres no padrão, de tal forma a tornar a equação simbólica entre o sujeito e o padrão verdadeira.

Em Gleam, o casamento de padrões é um dos principais meios utilizados para acessar os campos de tipos definidos pelo usuário, além de servir como mecanismo de controle de fluxo. Por exemplo, por meio dela, é possível determinar a qual variante uma instância de um tipo definido pelo usuário pertence, e associar os seus campos a variáveis que podem ser usadas, então, no interior do consequente da cláusula.

Para implementar o casamento de padrões, podemos encarar um padrão como uma tripla contendo: um conjunto de checagens que determinam se o padrão casa; um conjunto de atribuições que devem ser realizadas caso o padrão case; e um conjunto de padrões aninhados em seu interior. Desta forma, uma abordagem simples para realizar a geração de código para o casamento de padrões emerge. Primeiro, emita código que verifique se o padrão case. Se ele não casar, pule para a próxima cláusula; caso contrário, emita o código que realize as atribuições. Faça o mesmo recursivamente com todos os subpadrões até que todos eles sejam processados. Se, no fim, o fluxo de execução não saltou para fora, isso quer dizer que o padrão casou, e a expressão consequente da cláusula deve ser avaliada e retornada na pilha. Um exemplo de uma expressão `case` traduzida para Wasm pode ser visto na Listagem 15. Note que após a sequência de blocos gerados para os padrões, inserimos uma instrução `unreachable`. Esta instrução existe para garantir ao verificador Wasm que os tipos do bloco estão corretos, mas como o Gleam exige exaustividade no casamento de padrões, essa instrução nunca é alcançada. Obrigatoriamente, um dos blocos será selecionado e este saltará para fora do bloco externo.

Concretizamos essa implementação no compilador dividindo-a em três partes: uma que lida com a estrutura de blocos do casamento de padrões; outra que compila os padrões para representações em alto-nível das verificações e atribuições; e, por fim, uma que converte essas representações em alto-nível para código Wasm. Para cada cláusula, nós instanciamos um novo ambiente de referenciamento ligado ao atual, estabelecemos a estrutura de blocos e realizamos a compilação, tradução e costura do código gerado por cada expressão na cláusula.

A linguagem Gleam dá suporte a vários tipos de padrões, incluindo literais inteiros e de ponto flutuante, reatribuições, criação de variáveis, valores coringa, uso de construtores

de tipos, literais de strings, prefixos de strings, literais de *BitArrays*, variáveis de *BitArrays* e desestruturação de listas e tuplas. Apenas os padrões relacionados a BitArrays não foram implementados.

```
fn por_extenso() {
  let x = 1
  case x {
    1 -> "um"
    _ -> "não sei"
  }
}
```

Casamento de padrões em Gleam

```
(module
  (type $String
    (array (mut i8)))
  (type $fun@por_extenso
    (func (result (ref $String))))
  ;; ...
  (func $por_extenso (type $fun@por_extenso) (result (ref $String))
    (local $"#local0 #assign#0" (@name "#assign#0") i32)
    (local $x i32)
    (local $"#local2 #case#2" (@name "#case#2") i32)

    ;; let x = 1
    ;; (as linhas adicionais são devido ao
    ;; casamento de padrões da atribuição)
    i32.const 1
    local.set $"#local0 #assign#0"
    local.get $"#local0 #assign#0"
    local.set $x
    local.get $"#local0 #assign#0"
    drop

    ;; case x {
    ;; aqui nós obtemos o valor do sujeito e armazenamos
    ;; numa local
    local.get $x
    local.set $"#local2 #case#2"

    block (result (ref $String)) ;; label = @1
      block ;; label = @2
        ;; é igual a 1?
        i32.const 1
        local.get $"#local2 #case#2"
        i32.eq
        i32.eqz
```

```
35          ;; se não for, saímos deste bloco
36          ;; e vamos para o próximo
37          br_if 0 (;@2;)
38          ;; retorna a string "um"
39          i32.const 0
40          i32.const 2
41          array.new_data $String 11
42          br 1 (;@1;)
43        end
44        block ;; label = @2
45          ;; padrão coringa, sempre verdadeiro
46          i32.const 1
47          i32.eqz
48          br_if 0 (;@2;)
49          ;; retorna a string "não sei"
50          i32.const 0
51          i32.const 8
52          array.new_data $String 12
53          br 1 (;@1;)
54        end
55        ;; inalcançável; casamento de padrões é
56        ;; exaustivo
57        unreachable
58      end
59    )
60    ;; ...
61    (data (;11;) "um")
62    (data (;12;) "n\c3\a3o sei")
63  )
```

Listagem 15 – Casamento de padrões em Wasm

Dessas formas de padrões, a mais interessante é o de construtores de tipos. É com esse padrão que podemos determinar a qual variante um valor de um tipo pertence. Para fazermos isso, em tempo de execução, carregamos o discriminante da estrutura na pilha e checamos se o seu valor é igual ao discriminante esperado naquele padrão. O construtor possui um subpadrão para cada campo do tipo, que devem ser avaliados e atribuídos recursivamente no caso da variante externa casar com o sujeito. Quando constatamos que o sujeito casa, emitimos uma instrução explícita de conversão de referência Wasm, necessária para o módulo Wasm ser validado, e os campos do subtipo da variante acessados. O modo como passamos e armazenamos esses valores intermediários no casamento de padrões é via variáveis locais *Wasm*.

As atribuições realizadas por um padrão são retornadas no tipo que representa o padrão traduzido, e as associações encontradas são inseridas no ambiente de referenciamento do padrão, antes da emissão da expressão consequente.

Além das expressões `case`, Gleam permite o uso de casamento de padrões nas atribuições (sentenças `let`). Neste último caso, a mesma infraestrutura para a expressão `case` é utilizada, mas nenhuma expressão é avaliada após o casamento; o código gerado para no último conjunto de atribuições. O código gerado no `let` e no `let assert` é essencialmente o mesmo, com a diferença sendo que a expressão de verificação de casamento não é emitida no caso do `let` (pois a etapa de análise semântica garante estaticamente que o padrão casa).

É importante notar que o casamento de padrões não é a única forma de acessar membros de TDUs. A sintaxe de acesso direto a membros comuns permite que campos de mesmo nome, tipo e posição em todas as variantes de um TDU sejam acessados diretamente pelo seu nome, através do uso de um ponto (`.`) seguido do nome do campo. O modo como isso é viabilizado é explicado anteriormente nesta seção, e exemplificado de forma simplificada na Listagem 16.

```
1 type X {
2   X(x: Int, y: Float)
3   Y(x: Bool, y: Float)
4 }
5
6 fn main() {
7   let z = X(1, 2.0)
8   z.y
9 }
```

Acesso direto a membro

```
(module
  (type $sum@X
    (sub (struct (field i32) (field f64))))
  (type $typ@X.X
    (sub final $sum@X (struct (field i32) (field f64) (field i32))))
  (type $new@X.X
    (func (param i32 f64) (result (ref $sum@X))))
  (type $typ@X.Y
    (sub final $sum@X (struct (field i32) (field f64) (field i32))))
  (type $new@X.Y
    (func (param i32 f64) (result (ref $sum@X))))
  (type $fun@main (func (result f64)))
  (func $main (type $fun@main) (result f64)
    (local $"#local0 #assign#0" (@name "#assign#0") (ref $sum@X))
    (local $z (ref $sum@X))

    ;; let z = X(1, 2.0)
    i32.const 1
    f64.const 0x1p+1 (;=2;)
    call $new@X
    local.set $"#local0 #assign#0"
    local.get $"#local0 #assign#0"
    local.set $z
    local.get $"#local0 #assign#0"
    drop

    ;; x.z
    local.get $z
    ref.cast (ref $sum@X)
    struct.get $sum@X 1
  )
  (func $new@X (type $new@X.X) (param i32 f64) (result (ref $sum@X))
    i32.const 0
    local.get 1
    local.get 0
    struct.new $typ@X.X
  )
  (func $new@Y (type $new@X.Y) (param i32 f64) (result (ref $sum@X))
    i32.const 1
    local.get 1
    local.get 0
    struct.new $typ@X.Y
  )
)
```

Listagem 16 – Acesso direto a membro em Wasm

Compilando sequências de sentenças e funções, constantes

Uma função Gleam contém uma sequência de sentenças, cada qual gera um valor na pilha quando executada. Como uma função pode possuir mais de uma sentença em seu corpo, o gerador de código intercala entre cada sentença uma instrução *drop*, que remove o valor gerado pela expressão anterior do topo da pilha do Wasm. Ao fim da sua execução, resta apenas o valor gerado pela última sentença da função, e em Wasm este é definido como o valor de retorno da função. Sequências de sentenças também podem aparecer em blocos, os quais são outro tipo de expressão da linguagem.

A compilação de funções segue os princípios estabelecidos no restante desta seção. Primeiro, criamos um novo ambiente de referenciamento e adicionamos a ele os parâmetros da função, e instanciamos uma tabela de variáveis locais. Depois, geramos o código dos comandos presentes no seu corpo, atualizando o ambiente e o gerador de locais incrementalmente. Armazenamos dados sobre os parâmetros, tipo de retorno, variáveis locais e instruções geradas em uma estrutura, a qual é adicionada a uma lista de funções compiladas do módulo. Cada função recebe um índice, pelo qual ela pode ser chamada.

Chamadas a funções, como já mencionado, são compiladas avaliando-se os argumentos passados e invocando-se a função associada àquele nome. Implementamos dois casos de chamada de função. O primeiro é quando uma função declarada globalmente no módulo é chamada diretamente. Neste caso, emitimos uma instrução `call` com o índice da função diretamente. O outro caso é quando uma referência a uma função, armazenada em uma variável ou computada como resultado de uma expressão, é chamada. Nesta situação, produzimos uma referência àquela função e usamos a instrução `call_ref`, passando a assinatura da função sendo invocada. Fazemos essa distinção como uma otimização do caso mais comum, que é a chamada direta de funções.

Constantes em Gleam são declarações ao nível de módulo cujos valores não podem ser alterados. Para compilá-las para Wasm, criamos um global Wasm do tipo da constante e a inicializamos na função do módulo responsável por isso, a `start`. Construímos a função concatenando as rotinas de inicialização de todas as constantes declaradas no código, inclusive das instâncias únicas das variantes sem argumentos de TDUs.

Utilizando a API WASI

Em nosso gerador, como prova de conceito, adicionamos uma única função de saída que imprime um string na tela, a função `string_write_out`. A função utiliza a API `fd_write` do WASI 0.1, que recebe uma ou mais sequências de bytes e as escreve num descritor de arquivo especificado na pilha, que pode ser a saída padrão, por exemplo.

A API espera que a sequência a ser escrita esteja em uma memória linear, um tipo de item que pode ser declarado num módulo Wasm e que constitui um arranjo linear de

bytes que podem ser acessados e manipulados. Contudo, a nossa representação adotada para as strings, o tipo `array` da proposta Wasm GC, não utiliza a memória linear e não permite que passemos diretamente os dados da string ao WASI. Para resolver este problema, realizamos uma cópia temporária da string para a memória linear e então invocamos a função `fd_write` para que ela escreva a cadeia na tela.

Para acessar a função `string_write_out`, o usuário precisa declarar uma função externa do tipo `wasm`, com módulo `"gleam"` e nome do membro `"string_write_out"`. A Listagem 17 mostra um programa simples que permite que uma string seja impressa na saída padrão, quando executado.

```
1 @external(wasm, "gleam", "string_write_out")
2 fn print(str: String) -> Nil
3
4 pub fn main() {
5   print("Hello, world!\n")
6 }
```

Listagem 17 – Exemplo de código que escreve na saída padrão

Para auxiliar no uso deste recurso, implementamos uma função de nome `"int_to_string"` e começamos a planejar uma função `"float_to_string"`, que convertem inteiros e números de ponto flutuante para `Strings`, respectivamente.

No futuro, pretendemos disponibilizar meios mais diretos de acessar outras API WASI, para que eventualmente consigamos implementar a biblioteca padrão da linguagem em Wasm.

3.5 Serialização do módulo Wasm

Com essas regras de tradução, ao fim do processo, teremos um conjunto de funções, tipos e constantes gerados a partir do módulo processado. Resta-nos construir o módulo Wasm resultante contendo estas informações e salvá-lo em disco, num processo chamado de serialização. Fazemos este processo com o auxílio da biblioteca Rust*wasm-encoder*[16], que fornece uma interface de baixo-nível para codificação de arquivos Wasm no formato binário.

A construção de um módulo se dá por seções, que devem ser declaradas segundo uma ordem especificada na documentação Wasm. Primeiro, emitimos a seção de tipos, contendo tipos para todas as funções e para todos os tipos definidos pelo usuário no módulo Gleam e seus construtores. Depois, marcamos que estamos importando a função `fd_write` da API WASI 0.1. Em seguida, as declarações das funções e dos construtores

[16] *wasm-encoder*. Disponível em: <https://crates.io/crates/wasm-encoder>

em si são emitidas na seção das funções, referenciando os tipos declarados anteriormente, e geramos índices para a função `start`. Adicionamos a memória linear temporária que utilizamos com a função de saída. Caso alguma variável global tenha sido instanciada, elas também são adicionadas a uma seção contendo as globais. As funções Gleam públicas são marcadas para exportação pelo módulo. A função de inicialização das globais é definida e marcada como tal. O código das funções é armazenado na seção de código, referenciando os índices declarados na seção de funções e de tipos. Em seguida, registramos as seções `data` contendo as literais dos strings do módulo e, por fim, gravamos os nomes de todas essas entidades em uma seção especial de nomes, para facilitar a leitura e depuração do código Wasm. O módulo é gerado e um arquivo com extensão *.wasm* é salvo no sistema de arquivos virtual, e isto conclui a nossa implementação do gerador de código.

4 Análise e avaliação

Nesta seção, discutimos o nosso trabalho no contexto dos objetivos estabelecidos na Seção 1, mencionamos as limitações do gerador desenvolvido e comentamos sobre as principais dificuldades encontradas durante o desenvolvimento deste trabalho. Indicamos também algumas perspectivas para trabalhos futuros e realizamos uma comparação com trabalhos relacionados na área.

4.1 Avaliação dos objetivos

Nosso protótipo consegue produzir código Wasm para um subconjunto considerável da linguagem Gleam, que pode ser suficiente para o ensino básico de linguagens de programação. Neste processo, estabelecemos um mapeamento entre a maioria dos recursos das linguagens Gleam e Wasm, utilizando Wasm GC, com um caminho claro para a implementação dos recursos pendentes.

Ao todo, nossa implementação introduziu/alterou cerca de seis mil linhas de código (6 kLOC[17]) e foi desenvolvida no prazo de cinco meses e uma semana. Devido ao tempo limitado, nem todos os recursos da linguagem puderam ser implementados até o prazo de entrega deste trabalho.

Finalmente, o documento aqui produzido contém um relato e descrição do processo de construção do gerador de código e um detalhamento do processo de mapeamento das principais características da linguagem Gleam. O texto também contém perspectivas de melhorias futuras, conforme será detalhado posteriormente nesta seção.

4.2 Dificuldades

Desenvolvemos o nosso protótipo de maneira incremental, ao passo que estudávamos e aprendíamos técnicas de como fazer. Neste processo experimental, enfrentamos algumas dificuldades devido à relativa novidade da área a nós. Descrevemos nesta seção alguns dos maiores desafios encontrados.

Strings e o Wasm GC

Em sua revisão à época do desenvolvimento deste projeto, o Wasm GC não previa um tipo próprio para strings, tendo que ser utilizado um tipo análogo tal como o `array` para armazenar este tipo de informação. Contudo, a existência de APIs extensas que utilizam memória linear e a falta de um método canônico de converter objetos GC nas representações comuns dificultaram a implementação de strings neste projeto. Com isto em vista, precisamos escrever manualmente várias funções Wasm para dar suporte a operações

[17] *kilolines of code* – kilolinhas de código.

de strings presentes em Gleam, como igualdade, concatenação, *substring* e cópia para memória linear. Existem propostas para adicionar um tipo string formalmente ao Wasm GC, mas elas ainda estavam em discussão à época do trabalho.

Falta de uma especificação formal da linguagem Gleam

A linguagem Gleam é nova e ainda parcamente documentada. Durante a implementação, descobrimos algumas construções sintáticas interessantes que não sabíamos existirem pela documentação oficial. A falta de uma especificação formal nos fez assumir algumas propriedades a respeito de como a linguagem deveria ser compilada e executada. Nossa implementação não é completamente compatível com a implementação Erlang nem com a JavaScript, mas na ausência de parâmetros para medir o quão conformante um gerador de código é (além da suíte de testes disponível no repositório do compilador), não temos critérios objetivos para julgar se nossa implementação é válida ou não.

Integrando a implementação no compilador existente

O compilador Gleam é grande e muitas funções e tipos dele não estão documentados, embora esteja relativamente bem-estruturado. Os pontos que se relacionam com a escolha do gerador não estão concentrados em um único lugar. Pela estrutura e organização do código, é evidente que o compilador não foi originalmente projetado para acomodar mais de um gerador, com a intenção original sendo apenas gerar código Erlang. Teorizamos que a adição do JavaScript foi feita na medida do possível, adequando-se à estrutura existente, e ainda é possível observar alguns traços dessa transição na sua organização interna.

Tivemos que adicionar uma nova variante a um tipo em especial, a enumeração `Target`, para adicionar nosso gerador ao compilador. Esta enumeração continha dois membros, Erlang e JavaScript. Para determinar os locais que necessitariam ser modificados, inspecionamos os locais onde o tipo era utilizado e usamos as mensagens de erro do Rust para nos auxiliar. Como Rust é uma linguagem estaticamente tipada e as suas expressões `match` (análogas às expressões `case` em Gleam) são exaustivas, o compilador informa todos os lugares com expressões `match` que não estavam tratando a nova variante `Wasm`. Inicialmente, preenchemos esses lugares com a macro `todo!()`, que satisfaz o verificador de tipos do compilador e permite a compilação do código, mas emite um pânico em tempo de execução caso alcançada, e depois retornamos a elas conforme nossa implementação foi avançando.

No total, modificamos partes da análise sintática, da representação de tipos e da análise semântica para acomodar a presença de implementações de tipos e funções externas Wasm; e adicionamos o novo alvo de compilação Wasm à ferramenta de linha de comando do Gleam. Esperamos que, com tempo, o compilador torne-se mais bem documentado e que intervenções como a nossa se tornem mais simples.

4.3 Limitações

Tipos restantes da linguagem

Um dos recursos mais importantes que não foi incluso em nossa implementação é o suporte a tipos de dados e funções genéricas. Um tipo genérico permitem que usuário o usuário declare "variáveis de tipo", que podem ser associadas aos campos do tipo. Já as funções genéricas permitem ao usuário declarar variáveis de tipo para os seus parâmetros ou saída. Assim como uma função pode ser declarada uma vez e chamada com parâmetros diferentes, tipos e funções genéricas podem ser declarados uma vez e utilizados com diferentes tipos concretos.

Como um exemplo, o Gleam fornece os tipos `List(a)` e `Result(value, error)`. Observe que eles possuem, entre parênteses, uma lista de variáveis de tipo: o `List` possui uma variável `a`, que é o tipo dos elementos dessa lista, e o `Result` possui as variáveis `value` e `error`, sendo os tipos dos valores internos do caso `Ok` e do caso `Error`, respectivamente. Essa definição genérica permite que falemos dos conceitos de um tipo resultado e de uma lista de maneira abstrata, e implementemos funções que sirvam para todos os tipos de listas e de tipos resultado. Não precisamos definir tipos `ListInt`, `ListFloat`, `ListString`; uma definição genérica basta.

Existem duas principais estratégias utilizadas para implementar tipos genéricos, também chamado de polimorfismo paramétrico. A primeira é a remoção dos tipos (*type erasure*), onde, no momento da compilação, uma única cópia da função ou tipo é gerada, substituindo todas as variáveis de tipo por um supertipo comum. Esta abordagem é utilizada na linguagem Java, onde os valores genéricos são tratados como instâncias de `Object`, a superclasse de todas as classes, e conversões para os tipos reais são inseridos pelo compilador onde necessário.

Esta abordagem é segura porque a remoção dos tipos ocorre apenas após a verificação de tipos pelo compilador. Contudo, esta abordagem tem desvantagens: a primeira é que toda informação sobre os tipos usados na parametrização são perdidas em tempo de compilação, e a segunda é que este esquema requer um mecanismo de despacho dinâmico para métodos, o que adiciona um *overhead* de desempenho. No caso do Java, existe um aspecto negativo adicional: tipos primitivos como `int`, `float` ou `boolean` não herdam de `Object`, e assim não podem ser usados como parâmetros de tipos.

A segunda estratégia é a monomorfização, na qual uma cópia não genérica da função ou tipo é gerada para cada combinação de parâmetros de tipo utilizada. A principal vantagem é que a cópia gerada usa os tipos especificados diretamente, eliminando a necessidade de despacho dinâmico. Por outro lado, cada instanciação gera uma cópia do código diferente que precisa ser analisada, compilada e salva, aumentando o tempo de compilação e o tamanho dos binários gerados. Esta abordagem é usada em linguagens

como C++ e Rust.

Em versões futuras, pretendemos utilizar a estratégia de monomorfização em nossa implementação de tipos genéricos. Faríamos isso buscando um melhor desempenho em troca de binários ligeiramente maiores, e também porque os tipos Wasm GC que podem ser referenciados não incluem os tipos primitivos, como `i32` e `f64`. Se utilizássemos remoções de tipos, enfrentaríamos os mesmos problemas do Java. Para possibilitar isso, precisaríamos reformar o modo como as entidades são geradas e inseridas na tabela de símbolos para que isso pudesse ser feito sob demanda. Isso é necessário pois a primeira vez que uma instanciação de um tipo genérico ocorre pode ser apenas no interior de uma função, as quais atualmente só percorremos após termos finalizado a declaração e geração de todos os tipos da tabela de símbolos.

Este mecanismo nos permitiria implementar de maneira simples os tipos `Result`, `List` e tuplas (estas são adicionalmente genéricas no número de valores que podem possuir). Com estes tipos implementados, poderíamos então adicionar os padrões que faltam relacionados a eles ao nosso sistema de casamento de padrões. Além disso, poderíamos também adicionar apelidos de tipos (*type aliases*), que associam um novo nome a um tipo existente, o qual pode ser um tipo concreto, um tipo genérico ou um tipo concreto construído a partir de um tipo genérico.

Uma otimização que poderíamos fazer para os casos em que as variantes de um TDU não possuem campos seria abandonar a representação de *struct* e utilizar apenas um inteiro, representando o discriminante; deste modo evitaríamos a indireção de uma referência a um objeto Wasm GC. Isto já acontece com os tipos `Bool` e `Nil`, mas esse recurso generalizaria a otimização para todos os tipos com essa característica. Outra otimização semelhante que poderia ser feita é o uso do tipo `i31`, um inteiro de trinta e um bits, que pode ser passado em qualquer lugar onde uma referência é esperada em Wasm. Esse tipo poderia ser aproveitado nos TDUs que possuem variantes com campos e sem campos; as variantes sem campos poderiam ser passadas apenas como os valores dos seus discriminantes no formato `i31`, e as variantes com campos seriam passadas como referências a `structs` normalmente. Isso eliminaria um acesso à memória nos casos em que as variantes são simples, sem campos, melhorando o desempenho. O tipo `i31` é codificado como um inteiro de trinta e dois bits, mas um deles é reservado para diferenciá-lo das outras referências.

Outro tipo que não implementamos é o `BitString`, um tipo importante no ecossistema Erlang. `BitStrings` permitem que dados binários sejam construídos e processados de maneira declarativa, de um modo ergonômico e fácil. Para este recurso, usaríamos um `array` de bytes do Wasm GC. O arranjo de bytes seria tratado, então, com operações bit-a-bit de inteiros disponíveis no Wasm. Os padrões relacionados a `BitArrays` seriam implementados com essa representação em mente.

Suporte a projetos com mais de um módulo

Atualmente, nosso protótipo compila os módulos de um projeto individualmente e não possui capacidade de ligá-los em um único módulo completo. Isto significa que apenas projetos Gleam desenvolvidos em um único arquivo podem ser convertidas para um módulo Wasm por nosso gerador de código.

Existem diversos modos de implementar este recurso. Um deles, e o mais simples, é através da ligação manual dos módulos. O código receberia uma lista de módulos e os concatenaria, ajustando os índices relativos nas instruções apropriadamente. Seria necessário avaliar como alguns recursos como a cópia de `Strings` para a memória linear interagiriam com mais de um módulo, e também precisaríamos desduplicar as definições comuns dos dois módulos.

Outra abordagem, mais interessante, é aproveitar o modelo de componentes Wasm (Bytecode Alliance, 2024a) e compilar cada módulo como um componente Wasm. Isto permitiria o uso de ferramentas já existentes para combinar vários componentes em um único módulo Wasm, além de facilitar a integração com outras bibliotecas que adotam o modelo. Contudo, o modelo de componentes Wasm atualmente não oferece suporte ao Wasm GC, visto que ambos as propostas estão em desenvolvimento e não foram incorporadas oficialmente à especificação Wasm. O modelo de componentes define um conjunto de regras de transformação de vários tipos de dados para uma representação comum, permitindo o compartilhamento deles entre módulos, mas não especifica mapeamentos para os tipos definidos pelo Wasm GC. Mesmo que houvesse um mapeamento destes tipos, precisaríamos converter as definições dos tipos Gleam para o sistema de tipos do modelo de componentes. Assim, esta abordagem depende de futuras atualizações do padrão Wasm.

Com este recurso, também poderíamos importar itens de outros módulos, um recurso usado virtualmente em todo projeto Gleam. É importante também que o sistema forneça suporte à importação de funções externas de módulos Wasm arbitrários, o que permitiria que recursos fossem implementados em outras linguagens de programação e então usados em Gleam com facilidade.

Fechamentos (*closures*)

Outro recurso importante que ainda desejamos implementar em nosso projeto é a declaração de funções anônimas que capturam o seu ambiente, os fechamentos (*closures*) (SEBESTA, 2011), que é um recurso comum em linguagens de programação funcionais e que aumenta a expressividade da linguagem.

Um fechamento é criado quando uma função usa variáveis do seu escopo local que não foram declaradas dentro dela. Tais variáveis são chamadas de variáveis livres. Quando essa função é retornada ou armazenada em uma variável, é necessário que as variáveis livres

da função sejam movidas e armazenadas no *heap* do Wasm GC, para que ela não acesse referências inválidas na pilha quando a função for chamada. Em nossa implementação, precisaríamos analisar quais funções anônimas são fechamentos e quais variáveis são livres no corpo da função. Além disso, precisaríamos definir como as variáveis capturadas seriam acessadas pela função, o que provavelmente envolveria adicionar um novo parâmetro aos fechamentos que referenciaria as variáveis guardadas no *heap*.

Por fim, isso também requereria a capacidade de declarar e gerar funções sob demanda, pois funções anônimas podem ser declaradas em qualquer lugar no interior de uma função. Este recurso ficaria dependente de uma futura reestruturação do projeto.

Outros itens da linguagem restantes

Um recurso simples que poderia ser implementado de maneira simples, mas para o qual não tivemos tempo, é o das expressões `panic` e `todo` com mensagens de erro. Já temos rotinas que escrevem mensagens de texto na tela e bastaria conectar ambos os recursos.

Outros itens na lista de recursos futuros são as estruturas sintáticas de *pipelines* e de atualização de TDUs. Um *pipeline* é uma estrutura sintática que permite expressar composição de funções de maneira concisa. Em Gleam, ao invés de escrever uma invocação como `a(b(c(True),0),1))`, podemos usar o operador *pipeline* para escrever a expressão sintaticamente equivalente `c(True) |> b(0) |> a(1)`. A etapa de análise do compilador já coleta todas as informações necessárias para realizar esta transformação. Já a sintaxe de atualização de TDUs permite que novas instâncias de um TDU sejam criadas contendo, por padrão, os valores de outra instância do mesmo tipo, com a possibilidade de alterar um ou mais campos para outro valor. Isto é um recurso ergonômico importante, pois valores em Gleam são imutáveis, e essa sintaxe facilita o caso em que um único campo de uma instância precisa ser trocado. A Listagem 18 mostra um exemplo dessa sintaxe em ação.

```
1  import gleam/io
2
3  pub type Struct {
4    Struct(x: Int, y: Float, z: Bool)
5  }
6
7  pub fn main() {
8     let a = Struct(x: 1, y: 2.0, z: True)
9     let b = Struct(..a, y: 3.0)
10    io.debug(b) // Struct(x: 1, y: 3.0, z: True)
11 }
```

Listagem 18 – Exemplo de sintaxe de atualização de registro

Uma otimização importante que não realizamos, mas que planejamos no futuro, é a otimização de chamadas em posição de cauda. Uma chamada está em posição de cauda quando ela é a última ação realizada no interior de uma função, e o seu valor é retornado pela função. Nestes casos, a chamada em cauda não precisa ser empilhada sobre a função chamadora na pilha; o quadro da função chamadora na pilha pode ser substituído pelo da função chamada, e a semântica de execução permaneceria a mesma. Em Wasm, as instruções `return_call`,`return_call_indirect` `return_call_ref`, que desempilham a função atual antes de realizar a chamada seguinte. Esta otimização é importante para linguagens de paradigma funcional, pois elimina o problema de estouros da pilha em um contexto onde vários algoritmos são expressos de maneira recursiva.

Por fim, pensamos em implementar inteiros com precisão arbitrária. Isto adicionaria bastante complexidade ao projeto e acrescentaria um *overhead* de desempenho a todas as operações com inteiros, mas seríamos mais compatíveis com o ecossistema Erlang.

A biblioteca padrão

A linguagem Gleam tem uma biblioteca padrão com vários recursos importantes, como funções de saída na tela, mapas, dentre outros. Ela pode ser utilizada em uma aplicação Gleam adicionando-a como uma dependência na configuração do projeto. Embora relativamente abrangente, uma porção grande dela é implementada externamente. Por exemplo, as funções do módulo `gleam/io` e `gleam/dynamic` não poderiam ser implementadas exclusivamente em Gleam, e requerem acesso a recursos do ambiente de execução.

O desenvolvimento, portanto, necessitaria de uma biblioteca de apoio, escrita em Wasm, que implementasse essas funções e tipos externos. Precisaríamos escrevê-la em Wasm ou em alguma linguagem que compila para Wasm, e depois ainda precisaríamos de suporte a múltiplos módulos.

Contudo, desenvolvendo-se a biblioteca padrão, conseguiríamos disponibilizar nossa implementação para a parte do ecossistema Gleam que utiliza apenas a linguagem e a biblioteca, o que ampliaria o alcance e aplicabilidade do nosso trabalho.

4.4 Trabalhos relacionados

A compilação de linguagens de programação é uma área tradicional e bem explorada de modo geral, e o nosso trabalho não é único no âmbito da implementação de geradores de código Wasm. Várias linguagens já possuem suporte a essa linguagem-objeto.

Rust é uma linguagem de programação multi-paradigma de propósito geral. A linguagem incorporou o alvo de compilação Wasm na sua edição de 2018, construído sobre a infraestrutura LLVM. Um gama de bibliotecas existe no repositório de pacotes da

linguagem que fornece suporte para desenvolvimento Wasm com Rust (Rust WebAssembly Working Group, 2024).

Outra linguagem de programação que possui um gerador de código Wasm é a linguagem Roc, uma linguagem puramente funcional projetada com sintaxe amigável e compilada para código nativo de alto desempenho (Roc Community, 2024). Os autores da implementação relataram que a implementação do gerador requereu menos infraestrutura do que os demais, em especial devido à natureza ligeiramente mais alto-nível da linguagem Wasm (CARROLL, 2021).

O compilador de C/C++ para JavaScript, Emscripten[18], possui uma retaguarda de geração de código WebAssembly. O Emscripten originalmente compilava código C/C++ para asm.js, um subconjunto da linguagem JavaScript projetado para rápida execução, mas após a conclusão da primeira versão do WebAssembly o projeto passou a emitir Wasm diretamente (ZAKAI, 2015). Emscripten usa a infraestrutura LLVM e o projeto Binaryen[19] na geração de código. Binaryen é uma biblioteca de infraestrutura de compilação de Wasm escrita em C++ (Contribuidores Binaryen, 2024).

Em especial, nosso trabalho não é o único que implementou um gerador de código Wasm para a linguagem Gleam. Maywood (2024) conduziu uma implementação de um gerador Wasm para um subconjunto reduzido da linguagem Gleam, assim como nós, e em seu relatório a autora relatou as técnicas utilizadas e fez um apanhado geral do processo de implementação e das dificuldades. A implementação dela foi uma das motivações por trás da realização do trabalho, a saber por alguns motivos.

Um dos motivos foi que a implementação de Maywood utiliza remoção de tipos ao invés de monomorfização para suas funções e seus tipos genéricos. Isso gerou consequências importantes para o perfil de desempenho do código. Em primeiro lugar, os tipos primitivos inteiros e de ponto flutuante foram todos empacotados em *structs* Wasm GC, o que significa que cada uso de um valor requer um acesso à memória. Em segundo lugar, devido à perda de informações sobre os tipos após a compilação, algumas funções genéricas não podem ser implementadas de maneira eficiente devido a uma possível ambiguidade entre os tipos. Considere uma função que recebe dois argumentos de um mesmo tipo genérico e realiza a operação de igualdade entre eles. Para realizar uma implementação correta, seria necessário testar as referências para todos os tipos possíveis em Wasm e decidir qual das implementações de igualdade seria selecionada. Estas consequências impactam muito o desempenho dos módulos compilados. Em nosso gerador, pretendemos implementar tipos e funções genéricas por monomorfização, o que nos permitiria alcançar um melhor desempenho nesse âmbito.

Além disso, em seu trabalho, a autora não implementou nenhum mecanismo de

[18] Emscripten. Disponível em: <https://emscripten.org/>
[19] Binaryen. Disponível em: <https://github.com/WebAssembly/binaryen>

entrada e saída. Nosso projeto realizou uma integração experimental com o padrão WASI, através da função implícita `string_write_out`, conforme descrito na Seção 3. Por fim, nosso trabalho corrige algumas limitações presentes no trabalho de Maywood, como, por exemplo, a adição de suporte para tipos mutualmente recursivos.

4.5 Alternativas

Além das abordagens apresentadas em trabalhos relacionados, o uso de um alvo WebAssembly não é o único caminho para atingir nossos objetivos de produzir executáveis com suporte multiplataforma, tamanho reduzido, segurança e de ligação estática para a linguagem Gleam. Outras linguagens de programação exploraram meios diferentes, alguns dos quais analisamos a seguir.

Uma alternativa seria implementar um gerador de código para um conjunto de instruções nativo ao invés do Wasm. Por um lado, o código gerado não possuiria nenhum nível de indireção e haveria mais oportunidades para realizar otimizações do código. Esta abordagem é comum em várias linguagens compiladas e interpretadas, como C, Lua (LuaJIT) e JavaScript. Contudo, esta opção possui uma complexidade de implementação maior do que a de um gerador de código Wasm, que está um nível de abstração acima das linguagens de montagem nativas, e a implementação suportaria apenas um único conjunto de instruções. Ademais, a implementação teria de ser especializada para cada sistema operacional manualmente, pois detalhes como as convenções de chamada variam de ambiente para ambiente, restringindo a portabilidade desta alternativa. Existem ainda desafios relacionados à segurança do código gerado, que teria que ser inspecionado e validado para garantir segurança de memória e outros aspectos: código Wasm, mesmo incorreto, não viola as proteções de segurança do ambiente de execução.

Outro caminho seria utilizar a infraestrutura de geração de código de um compilador já existente para gerar Wasm ou código de máquina diretamente, como as do LLVM ou do GCC. A linguagem de programação Rust usa este meio. O benefício desta abordagem é que esses compiladores já possuem várias otimizações implementadas, e o conjunto de recursos fornecidos por eles é rico e complexo. Contudo, estes projetos não são leves, e o seu uso os fixaria como dependências diretas do compilador Gleam, tornando-o muito mais complexo. Assim como na primeira alternativa, isso restringiria as plataformas da linguagem às suportadas pela infraestrutura, e embora o LLVM e o GCC possuam uma ampla gama de plataformas suportadas, quaisquer outras plataformas precisariam ser implementadas primeiro de forma genérica nessas infraestruturas antes de na linguagem Gleam. As mesmas ressalvas de segurança da geração de código nativo se aplicam aqui.

A transcompilação de Gleam para uma linguagem de alto nível, como C ou C++, é outra abordagem possível. Este método é utilizado por compiladores como o Cython. Como estas linguagens localizam-se em níveis de abstração próximos ao do Gleam, a tradução

seria mais direta, e aproveitar-se-ia a existência das ferramentas e compiladores existentes para estas linguagens. Entretanto, a tradução para estas linguagens não seria muito mais simples que a tradução para Wasm, pois o WebAssembly também possui estruturas de controle semelhantes a essas linguagens. A compilação para C ou C++ também tornaria o uso do Gleam em plataformas como navegadores Web mais difícil e requereria a compilação do código-fonte uma vez para cada plataforma desejada. Por fim, esta opção limitaria a linguagem às abstrações disponíveis nas linguagens-alvo, e forçaria a implementação a se contorcer para adequar-se às suas estruturas.

No todo, consideramos a opção por implementar um gerador de código Wasm mais adequada ao considerar-se os objetivos e restrições do projeto e os benefícios e contrapartidas das alternativas.

5 Conclusão

Neste trabalho, construímos um gerador de código Wasm para a linguagem de programação Gleam. Nossa implementação abrange um subconjunto considerável da linguagem e permite a sua utilização em contextos de ensino e de recursos restritos.

O gerador de código dá suporte a recursos como casamento de padrões, funções, atribuições, expressões, escopo léxico, tipos definidos pelo usuário, saída, a maioria dos operadores da linguagem, constantes e projetos com um único módulo, sem dependências. Utilizamos os recursos da linguagem WebAssembly e de sua proposta Wasm GC na implementação dessas porções da linguagem.

Nem todos os recursos da linguagem foram implementados. Alguns importantes, como fechamentos e tipos genéricos, precisam de uma reestruturação do código do gerador para serem implementados. Em retrospecto, com a informação de que a geração sob demanda de tipos e funções seria necessária, teríamos projetado a retaguarda de maneira diferente, e o desenvolvimento seria mais simples. A construção deste gerador de código foi uma grande experiência de aprendizado para nós e alguns erros que cometemos não seriam cometidos novamente nos próximos projetos.

Em trabalhos futuros, pretendemos adicionar os recursos remanescentes da linguagem ao gerador e assim completar o processo de tradução. Uma de nossas pretensões é contribuir a retaguarda ao repositório oficial da linguagem, para que mais usuários possam utilizá-la, e para isso, precisamos de uma cobertura total da linguagem de programação e de uma implementação completa da biblioteca padrão em Wasm. Também precisaríamos integrar mais efetivamente o gerador de código ao compilador e construir uma um conjunto robusto de testes de regressão.

Além disso, gostaríamos de explorar a compilação de aplicações Gleam em pacotes executáveis independentes, conforme mencionado na Seção 1, e consideramos que nossa implementação seja um marco importante nesse sentido. Desejamos também realizar uma análise comparativa de desempenho para validar a nossa abordagem em comparação com os demais geradores de código existentes no compilador, e com outros trabalhos. No mais, esperamos que o produto de nossos esforços até o momento represente uma contribuição significativa para a comunidade de usuários Gleam e para os interessados na construção de compiladores e no projeto e implementação de linguagens de programação.

Referências

AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas, 2ª edição*. [S.l.]: Editora Pearson, 2007. Citado 3 vezes nas páginas 8, 12 e 13.

ARMSTRONG, J. A history of erlang. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. [S.l.: s.n.], 2007. p. 6–1. Citado na página 8.

BARBOSA, M. A. L. *Algoritmos e Linguagens: Notas de Aula*. 2024. P. 23. Disponível em: <https://malbarbo.pro.br/arquivos/2024/6879/01-algoritmos-e-linguagens-notas-de-aula.pdf>. Citado na página 12.

Bytecode Alliance. *The WebAssembly Component Model*. 2024. Disponível em: <https://component-model.bytecodealliance.org/>. Acesso em: 13 dec. 2024. Citado 2 vezes nas páginas 25 e 53.

Bytecode Alliance. *WebAssembly Micro Runtime (WAMR)*. 2024. Disponível em: <https://github.com/bytecodealliance/wasm-micro-runtime>. Acesso em: 20 jul. 2024. Citado na página 11.

CARROLL, B. *Roc Language: Development backend for WebAssembly*. 2021. Disponível em: <https://github.com/roc-lang/roc/blob/b17996f6f772c1d9aead8f9175a1ca8e019b9d0f/crates/compiler/gen_wasm/README.md>. Acesso em: 20 jul. 2024. Citado na página 56.

CHOMSKY, N. Three models for the description of language. *IRE Transactions on information theory*, IEEE, v. 2, n. 3, p. 113–124, 1956. Citado na página 12.

Contribuidores Binaryen. *Binaryen*. 2024. Disponível em: <https://github.com/WebAssembly/binaryen>. Acesso em: 20 jul. 2024. Citado na página 56.

DOWEK, G. Higher-order unification and matching. In: _____. *Handbook of Automated Reasoning*. NLD: Elsevier Science Publishers B. V., 2001. p. 1009–1062. ISBN 0444508120. Citado na página 41.

GRIGORE, R. Java generics are turing complete. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 52, n. 1, p. 73–85, 2017. Citado na página 12.

HINDLEY, R. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, JSTOR, v. 146, p. 29–60, 1969. Citado 2 vezes nas páginas 8 e 15.

IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, p. 1–84, 2019. Citado na página 32.

MAYWOOD, D. *WebAssembly Target Implementation for Gleam*. Projeto final do curso de Ciência da Computação — Department of Computer Science, Loughborough University, May 2024. Acesso em: Orientador: Dr J. Day. Citado na página 56.

MILNER, R. A theory of type polymorphism in programming. *Journal of computer and system sciences*, Elsevier, v. 17, n. 3, p. 348–375, 1978. Citado 2 vezes nas páginas 8 e 15.

NYSTROM, R. *Crafting interpreters*. Genever Benning, 2021. Disponível em: <https://craftinginterpreters.com/>. Citado na página 13.

PERLIS, A. J. Special feature: Epigrams on programming. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 17, n. 9, p. 7–13, 1982. Citado na página 13.

PILFOLD, L. *Gleam*. 2024. Disponível em: <https://gleam.run>. Acesso em: 18 jul. 2024. Citado na página 15.

PILFOLD, L. *Gleam Language Tour*. 2024. Disponível em: <https://tour.gleam.run/everything/>. Acesso em: 20 jul. 2024. Citado na página 16.

Roc Community. *Roc Language Official Website*. 2024. Https://www.roc-lang.org/. Acesso em: 20 jul. 2024. Citado na página 56.

Rust WebAssembly Working Group. *The Rust and WebAssembly Book*. 2024. Disponível em: <https://rustwasm.github.io/docs/book/>. Acesso em: 20 jul. 2024. Citado na página 56.

SEBESTA, R. W. *Conceitos de Linguagens de Programação, 9ª edição*. [S.l.]: Bookman, 2011. Citado 5 vezes nas páginas 5, 8, 14, 17 e 53.

VALIM, J. *Elixir*. 2017. Palestra de abertura ministrada por José Valim, criador da linguagem Elixir, no evento ElixirConf EU 2017, em Barcelona, Espanha. Acesso em 6 dez. 2024. Disponível em: <https://www.youtube.com/watch?v=IZvpKhA6t8A>. Citado na página 8.

VELDHUIZEN, T. L. C++ templates are turing complete. *Available at citeseer. ist. psu. edu/581150. html*, Citeseer, 2003. Citado na página 12.

W3C. *GC Proposal for WebAssembly*. 2024. Disponível em: <https://github.com/WebAssembly/gc>. Acesso em: 1 set. 2024. Citado na página 23.

W3C. *WASI: Interfaces*. 2024. Disponível em: <https://wasi.dev/interfaces>. Acesso em: 20 jul. 2024. Citado na página 25.

W3C. *WebAssembly*. 2024. Disponível em: <https://webassembly.org>. Acesso em: 17 jul. 2024. Citado na página 21.

W3C. *WebAssembly: Security*. 2024. Disponível em: <https://webassembly.org/docs/security/>. Acesso em: 20 jul. 2024. Citado na página 22.

W3C. *WebAssembly Specification: Binary Format*. 2024. Disponível em: <https://webassembly.github.io/gc/core/binary/conventions.html>. Acesso em: 20 jul. 2024. Citado na página 22.

W3C. *WebAssembly Specification: Modules*. 2024. Disponível em: <https://webassembly.github.io/gc/core/syntax/modules.html>. Acesso em: 20 jul. 2024. Citado na página 21.

W3C. *WebAssembly Specification: Runtime Structure*. 2024. Disponível em: <https://webassembly.github.io/gc/core/exec/runtime.html>. Acesso em: 31 ago. 2024. Citado na página 36.

W3C. *WebAssembly Specification: Text Format*. 2024. Disponível em: <https://webassembly.github.io/gc/core/text/conventions.html>. Acesso em: 20 jul. 2024. Citado na página 22.

W3C. *WebAssembly System Interface (WASI)*. 2024. Disponível em: <https://wasi.dev/>. Acesso em: 20 jul. 2024. Citado na página 25.

W3C. *WebAssembly: Use Cases*. 2024. Disponível em: <https://webassembly.org/docs/use-cases/>. Acesso em: 20 jul. 2024. Citado na página 24.

YERGEAU, F. *UTF-8, a transformation format of ISO 10646*. [S.l.], 2003. Citado na página 32.

ZAKAI, A. *Compiling to WebAssembly: It's Happening!* 2015. Disponível em: <https://hacks.mozilla.org/2015/12/compiling-to-webassembly-its-happening/>. Acesso em: 20 jul. 2024. Citado na página 56.