Expansão de Suporte a Novas Linguagens de Programação na Ferramenta BenchGen

Projeto Orientado em Computação II

Matheus Marchesotti Dutra Ferraz
UFMG
Belo Horizonte, Brasil
matheusferraz@dcc.ufmg.br

Vinicius Francisco da Silva
UFMG
Belo Horizonte, Brasil
silva.vinicius@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brasil
fernando@dcc.ufmg.br

***Resumo*—A validação de compiladores e ferramentas de análise estática exige um vasto conjunto de programas de teste diversificados. A ferramenta BenchGen, baseada em *L-Systems*, sintetiza tais programas de forma determinística e escalável. Este trabalho, desenvolvido no contexto da disciplina Projeto Orientado em Computação II, deu continuidade à expansão do BenchGen, implementando suporte para quatro novas linguagens: D, Nim, Ada e Vale. O processo envolveu a adaptação da gramática gerativa e a implementação de módulos de geração de código específicos, com ênfase nos distintos modelos de gerenciamento de memória de cada linguagem. Os resultados demonstram que a ferramenta gera programas sintaticamente corretos, compiláveis e executáveis para D, Nim e Ada, com desempenho competitivo frente às referências em C e C++ — destacando-se Ada como a implementação mais eficiente do conjunto avaliado quando compilada sem as verificações de segurança em tempo de execução. Mediu-se cada linguagem com suas melhores opções de otimização (preferindo *back-ends* LLVM, como em C/C++ e no `ldc2` para D). Para Vale, discute-se a ausência de *ownership* compartilhado como um obstáculo fundamental à reutilização de estruturas, motivando uma estratégia alternativa baseada em um *pool* de reciclagem que, ainda assim, não atinge desempenho comparável.**

***Palavras-chave*—Compiladores, Fuzzing, Geração de Código, BenchGen, D, Nim, Ada, Vale**

I. Introdução

A complexidade dos sistemas de computação modernos, que incluem compiladores otimizadores, sistemas operacionais e bibliotecas de tempo de execução (*runtime*), torna a tarefa de validação e teste extremamente desafiadora. Ferramentas de *fuzzing* generativo, como o Csmith, têm sido fundamentais para encontrar defeitos (*bugs*) em compiladores C/C++. No entanto, o ecossistema de linguagens de programação de sistemas tem evoluído rapidamente com o surgimento de linguagens como Zig, V, Odin, Rust, D, Nim e Vale, criando uma demanda por ferramentas capazes de gerar *benchmarks* também para esses novos alvos.

O projeto BenchGen aborda essa lacuna utilizando Sistemas de Lindenmayer (*L-Systems*) para gerar programas de tamanho e complexidade arbitrários a partir de uma semente e regras de produção.

No Projeto Orientado em Computação I (POC I), o objetivo foi expandir o BenchGen para as linguagens V, Zig e Odin. Os resultados foram satisfatórios: a ferramenta passou a gerar programas sintaticamente corretos para esses alvos, a implementação de V transpilada para o *backend* C apresentou desempenho competitivo com C e C++, e as implementações em Zig e Odin demonstraram alta eficiência no gerenciamento de recursos, superando a implementação em C++ baseada em `std::vector`. Identificou-se ainda e corrigiu-se um gargalo de desempenho em Zig relacionado à escolha do alocador de memória.

O objetivo principal deste trabalho (POC II) foi dar continuidade a essa expansão, incluindo as linguagens **D**, **Nim**, **Ada** e **Vale**, conforme proposto inicialmente. Este relatório detalha as atividades de engenharia de software realizadas para integrar essas linguagens, os desafios sintáticos e semânticos — em especial os relativos aos diferentes modelos de gerenciamento de memória — e apresenta uma análise de desempenho dos *benchmarks* gerados.

II. Referencial Teórico

A. Geração de Benchmarks via L-Systems

A autossimilaridade é caracterizada pela semelhança de um padrão a parte de si mesmo, de forma que uma estrutura se repita em diferentes escalas. O BenchGen utiliza a propriedade de autossimilaridade em estruturas de controle de fluxo, como laços e condicionais, que podem ser modeladas recursivamente, para gerar programas. Um *L-System* é composto por um alfabeto, um conjunto de regras de produção e uma *string* inicial (axioma); a cada iteração, as regras são aplicadas recursivamente, produzindo sequências cada vez mais complexas. A ferramenta utiliza uma gramática livre de contexto estendida para sintetizar código que, embora gerado de forma estruturada, mantém semântica executável e determinística, permitindo a verificação de problemas de compilação por meio da comparação de *outputs* entre diferentes linguagens ou níveis de otimização.

B. Estrutura de Geração de Código

A geração de código no BenchGen baseia-se em dois tipos de construções. As de **Estrutura** definem o fluxo de controle do programa (`IF`, `LOOP` e `CALL`), enquanto as de **Comportamento** especificam a manipulação de dados (`new`, `insert`, `remove` e `contains`). O fluxo de execução dos programas gerados é controlado por uma variável `path`, cujos *bits* determinam o resultado dos desvios condicionais, permitindo explorar diferentes caminhos de execução de forma determinística a partir de uma semente.

C. Linguagens Alvo

- **D:** Linguagem de sistemas multiparadigma com sintaxe próxima a C/C++. Possui coletor de lixo (*garbage collector*) por padrão, mas permite gerenciamento manual de memória por meio da biblioteca C padrão.
- **Nim:** Linguagem de tipagem estática compilada para C, com gerenciamento de memória automático (ARC/ORC), mas que também oferece primitivas de alocação manual de baixo nível.
- **Ada:** Linguagem com forte ênfase em segurança e confiabilidade, amplamente utilizada em sistemas críticos. Emprega tipos de acesso (*access types*) e desalocação explícita controlada.
- **Vale:** Linguagem experimental cujo modelo de memória se baseia em *single ownership* com *generational references*, sem suporte a *ownership* compartilhado entre referências.

III. Metodologia e Desenvolvimento

A metodologia adotada seguiu o modelo iterativo e incremental estabelecido no semestre anterior: para cada linguagem alvo, repetiu-se o ciclo de implementação do gerador, teste de compilação e validação de execução. O principal eixo de dificuldade desta etapa foi a diversidade de **modelos de gerenciamento de memória** entre as linguagens, que exigiu adaptações específicas em cada gerador.

Uma decisão de projeto transversal a todas as linguagens é o uso de gerenciamento de memória **manual**, mesmo nas que dispõem de coletor de lixo. A motivação é de comparabilidade: o objetivo do BenchGen não é avaliar o desempenho de coletores de lixo, mas a execução da carga de trabalho gerada. Ao impor gerenciamento manual de forma uniforme — inclusive a mesma lógica de *pool* de arrays já consolidada nos *backends* de referência —, garante-se que a carga exercida seja equivalente entre os alvos e que linguagens com e sem coletor de lixo possam ser comparadas em igualdade de condições.

A. D

A linguagem D foi a primeira implementada nesta etapa. Seguindo a decisão de gerenciamento manual descrita acima, ainda que D disponha de um coletor de lixo, adotou-se o mesmo modelo de contagem de referências dos *backends* de C e C++: o gerador emite chamadas a `malloc`/`free` da biblioteca C padrão (`core.stdc.stdlib`), mantendo um campo de contagem de referências (`refC`) em cada estrutura e liberando os recursos quando essa contagem atinge zero. Cada função gerada é colocada em seu próprio módulo, com importações explícitas, e o ponto de entrada coordena as chamadas e o ciclo de vida das estruturas.

B. Nim

Para Nim, manteve-se a estratégia de alocação manual, contornando o gerenciamento automático (ARC/ORC) da linguagem. O gerador utiliza as primitivas de baixo nível `alloc`/`dealloc` e o tipo `UncheckedArray`, com conversões explícitas (`cast`), evitando os *seqs* idiomáticos e o coletor. A geração concentra todo o programa em um módulo principal que inclui os arquivos de função, exigindo atenção à ordem de definição dos símbolos — por exemplo, as rotinas do gerador de números pseudoaleatórios precisam ser definidas antes do módulo que calcula a variável `path`. A compilação utiliza `nim c -d:danger`, que ativa as otimizações de *release* e suprime as verificações em tempo de execução.

C. Ada

A implementação de Ada exigiu a adaptação ao seu sistema de tipos e à separação característica entre especificação (`.ads`) e corpo (`.adb`). O gerador emprega *access types* (`Array_T_Access`) para as estruturas dinâmicas e realiza a desalocação explícita por meio de `Ada.Unchecked_Deallocation`, instanciada em um pacote de tipos (`bench_types.ads`) gerado automaticamente. Parâmetros são passados por `access` e referenciados com o atributo `'Access`. A compilação utiliza `gnatmake` com otimização (`-O3`); avaliaram-se duas configurações quanto às verificações de segurança da linguagem — `safe` (verificações ativas, padrão) e `unsafe` (`-gnatp`, que as suprime) —, conforme discutido na Seção IV-D.

D. Vale

Vale apresentou o desafio conceitual mais profundo desta etapa. Seu modelo de memória baseia-se em *single ownership*: cada valor possui um único dono e não há *ownership* compartilhado entre referências. Essa restrição é incompatível com a estratégia adotada nos demais *backends*, na qual uma mesma estrutura (*array*) é compartilhada e reutilizada entre múltiplos escopos via contagem de referências.

Como alternativa, implementou-se um ***pool*** **de reciclagem de arrays** (*free-list*): em vez de compartilhar estruturas vivas, o gerador encaminha, por empréstimo (*borrow*), uma lista de arrays previamente liberados. A operação `takeOrMake` reutiliza um array do *pool* quando disponível ou aloca um novo; ao final de cada escopo, a rotina de liberação devolve os arrays ao *pool* (`pool.add`), exceto a variável de retorno. Trata-se do análogo, sob *single ownership*, do *pool* de arrays usado em C e Rust — com a diferença fundamental de que o *pool* só pode devolver estruturas que já foram liberadas, e nunca estruturas ainda vivas.

Apesar de funcionalmente correta, essa estratégia não elimina o custo imposto pela ausência de compartilhamento: como será discutido na Seção IV, os programas em Vale apresentaram desempenho significativamente inferior ao das demais linguagens. Por esse motivo, que também levou a integração de Vale a ainda não ter sido incorporada ao repositório oficial do projeto, Vale não foi incluída na comparação quantitativa de desempenho.

E. Integração Contínua

Todo o código desenvolvido para D, Nim e Ada foi submetido ao repositório oficial do projeto por meio de *Pull Requests*, mantendo o histórico de versionamento e revisão de código. O suporte a Vale permanece, por ora, em um *fork* pessoal.

IV. Resultados

A. Metodologia de Medição

A avaliação de desempenho foi conduzida em dois recortes complementares. A comparação por linguagem (Figura 2) foi executada na **máquina de referência do projeto**, idêntica à descrita no repositório de métricas (benchMetrics): um servidor com processador Intel® Xeon® E5-2680 v2 (2,80 GHz, 10 núcleos) e 32 GB de RAM, sob Ubuntu 20.04.6 LTS. O estudo de escalabilidade por profundidade (Figura 1) foi realizado em caráter preliminar em um notebook mais modesto, apenas para confirmar o crescimento do tempo de execução com o tamanho do programa; nesse recorte, sob carga sustentada, a maior profundidade torna-se limitada por alocação e largura de banda de memória, comprimindo as diferenças entre linguagens.

Em ambos os recortes, cada programa foi gerado a partir de um mesmo conjunto de regras de produção e semente, compilado com o *toolchain* nativo de sua linguagem — sempre com as **melhores opções de otimização disponíveis** — e executado múltiplas vezes, registrando-se o tempo de execução mediano (*wall-clock*). Buscou-se, sempre que possível, o uso de *back-ends* baseados em LLVM, em consonância com a configuração de referência: C e C++ com `clang -O3`; D com o compilador LLVM `ldc2 -O3 -release` (e não o `dmd`, de otimização menos agressiva); Julia, Zig e Odin já empregam LLVM internamente. As demais usaram suas configurações de *release* padrão: Nim com `-d:danger` e Go com `go build`. Para Ada, avaliaram-se duas configurações (`safe` e `unsafe`), discutidas na Seção IV-D; para C++, as variantes baseadas em *array* e em `std::vector`; e para V, ambos os *back-ends*, o C (`-prod`) e o nativo (`-b native`). Para Zig, em razão de incompatibilidades de API introduzidas na versão 0.16 do compilador, fixou-se a versão 0.14.1, na qual o código gerado compila sem alterações.

B. Corretude e Compilação

As implementações de D, Nim e Ada geraram, em todas as profundidades avaliadas, programas sintaticamente corretos, que compilaram e executaram sem erros. Confirma-se, assim, o principal objetivo do projeto: a ampliação da cobertura de linguagens do BenchGen com geração de *benchmarks* válidos.

C. Análise de Desempenho

A Figura 1 apresenta o tempo de execução em função da profundidade do programa (escala logarítmica), confirmando o crescimento monotônico da carga; a Figura 2 detalha a comparação de tempo entre as linguagens na máquina de referência, incluindo as variantes de Ada, C++ e V.

A análise dos resultados revela os seguintes comportamentos:

1) **Ada como destaque de eficiência:** na configuração `unsafe` (com supressão das verificações em tempo de execução, `-gnatp`), Ada apresentou o menor tempo de execução entre todas as linguagens avaliadas, situando-se à frente inclusive de C. Esse resultado é especialmente relevante por se tratar de uma das linguagens adicionadas neste trabalho: o gerenciamento explícito de memória via *access types* e `Unchecked_Deallocation`, compilado com otimização agressiva, produz código altamente eficiente. Já a configuração `safe`, que preserva as verificações, recai no grupo mais lento — diferença analisada na Seção IV-D.
2) **Referências competitivas:** logo após Ada `unsafe` aparecem C e a variante de C++ baseada em *array*, seguidas pelo *backend* C de V, por Zig e por Nim. Nim, em particular, posicionou-se à frente de Odin e D, confirmando que a estratégia de alocação manual contornando o coletor de lixo gera código competitivo com as referências de sistemas.

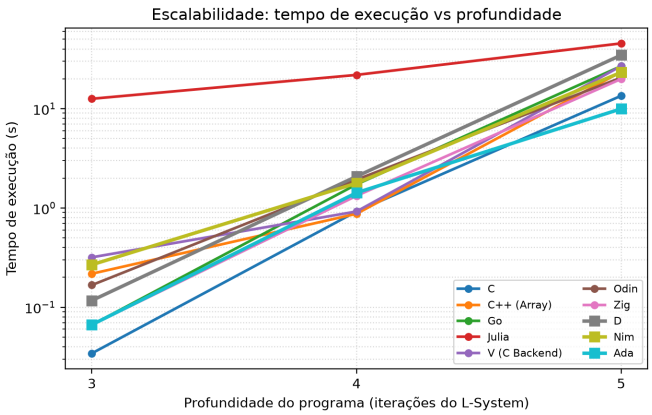

Figura 1. Tempo de execução (escala logarítmica) em função da profundidade do programa, para as linguagens com escalabilidade válida. As linguagens adicionadas neste trabalho (D, Nim, Ada) são destacadas com traço mais espesso e marcadores quadrados.

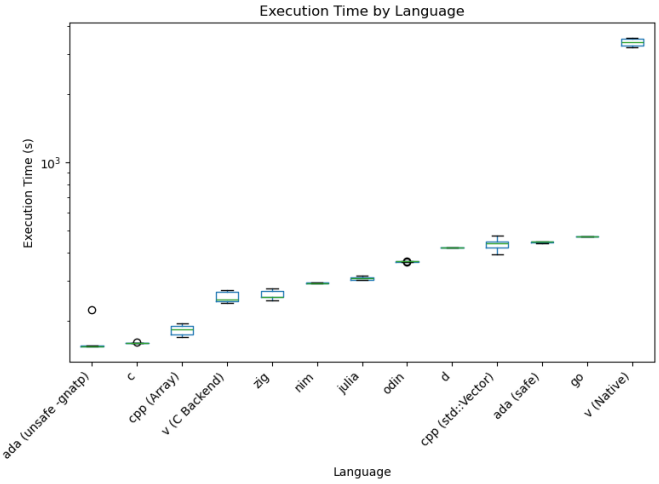

Figura 2. Tempo de execução por linguagem na máquina de referência (escala logarítmica), ordenado do mais rápido ao mais lento. Entre as novas linguagens do POC II estão D, Nim e Ada — esta última avaliada em duas configurações (`unsafe`, com verificações suprimidas, e `safe`). Mostram-se também as variantes *array* e `std::vector` de C++ e ambos os *back-ends* de V (C e nativo).

3) **D limitado pela alocação:** D figura entre as linguagens compiladas mais custosas. Verificou-se que o compilador não é o gargalo: o `ldc2` (*back-end* LLVM) produziu tempo praticamente idêntico ao do `dmd`, indicando que a carga é dominada pela alocação/liberação de memória, e não pela qualidade do código gerado.
4) **Julia amortizada:** ao contrário do que se observa em cargas menores, onde o custo fixo de inicialização (*JIT*) é dominante, na carga ampliada da máquina de referência Julia situou-se no meio do agrupamento, evidenciando que esse custo é diluído em execuções mais longas.
5) **C++ `std::vector` e Go:** a variante de C++ baseada em `std::vector` mostrou-se sensivelmente mais lenta que a baseada em *array*, à semelhança do já reportado no POC I; Go situou-se em faixa semelhante.
6) **V (nativo) vs. V (*backend* C):** reproduzindo o já observado no POC I, o *backend* nativo de V foi, com larga margem, o mais lento de todos — mais de uma ordem de grandeza acima das demais linguagens —, enquanto o *backend* C manteve-se agrupado com as linguagens de alto desempenho. A disparidade não decorre do código gerado, e sim do próprio *backend* nativo de V, ainda experimental, que carece das otimizações presentes no GCC/Clang utilizados pelo *backend* C.
7) **Escalabilidade:** todas as linguagens exibiram crescimento monotônico do tempo de execução com a profundidade (Figura 1), confirmando que os programas gerados exercitam, de fato, uma carga de trabalho proporcional ao tamanho do programa.

D. Ada: Safe versus Unsafe

Por padrão, o compilador de Ada insere verificações de segurança em tempo de execução a cada operação potencialmente perigosa: checagem de limites no acesso a *arrays*, verificação de referências nulas em *access types* e validação de faixas e restrições de tipo. Ao detectar uma violação, o programa levanta uma exceção (por exemplo, `Constraint_Error`) em vez de incorrer em comportamento indefinido — daí a segurança de memória que é a marca da linguagem. A configuração `unsafe`, compilada com `-gnatp`, suprime essas verificações; a `safe` (padrão) as mantém. Mediram-se ambas sobre exatamente a mesma carga de trabalho, de modo que a diferença reflita unicamente o custo das checagens.

O efeito no gráfico (Figura 2) é expressivo: a versão `safe` recai no grupo das mais lentas, enquanto a `unsafe` é a mais rápida de todo o conjunto. Isso torna explícito e mensurável o *trade-off* entre segurança e desempenho da linguagem, e indica que, para a comparação de eficiência reportada nas seções anteriores, a configuração `unsafe` é a referência adequada.

Mais notável é o fato de Ada `unsafe` superar inclusive C. Uma explicação provável está no sistema de tipos de Ada: suas regras de *aliasing* são mais restritivas que as de C e os *access types* são fortemente tipados, o que informa ao compilador que ponteiros distintos não se referem à mesma região de memória. Com essa garantia, o otimizador (de *back-end* GCC, comum ao GNAT) tem mais liberdade para reordenar e vetorizar acessos do que em C, onde o *aliasing* precisa ser tratado de forma conservadora. Vale a ressalva, contudo, de que a vantagem é estreita e pode refletir, em parte, variação de medição.

E. Limitações e Casos Excluídos

O único caso excluído da comparação quantitativa, por transparência metodológica, foi **Vale**: conforme discutido, a ausência de *ownership* compartilhado impede a reutilização de estruturas vivas. Mesmo com o *pool* de reciclagem implementado como alternativa, o desempenho permaneceu muito inferior ao das demais linguagens, o que — somado ao fato de a implementação ainda residir em *fork* — motivou sua exclusão dos gráficos.

Adicionalmente, registra-se uma ressalva de *toolchain* relativa a Zig: a partir da versão 0.16 o compilador removeu APIs da biblioteca padrão (tratamento de argumentos, alocadores, leitura de variáveis de ambiente) utilizadas pelo código gerado. Para preservar a fidelidade ao código de referência, as medições de Zig foram realizadas com a versão 0.14.1 do compilador. A atualização do *backend* de Zig para a API mais recente fica como trabalho futuro.

Esses casos ilustram um desafio recorrente em *benchmarking* de múltiplas linguagens: a sensibilidade a versões de *toolchain* e a fidelidade da carga de trabalho gerada entre *backends*.

V. Conclusão e Trabalhos Futuros

O objetivo da disciplina POC II foi atingido: o BenchGen passou a contar com suporte funcional para as linguagens D, Nim e Ada, gerando *benchmarks* sintaticamente corretos, compiláveis e com desempenho competitivo frente às referências — com destaque para Ada, a mais eficiente do conjunto em sua configuração com verificações suprimidas (`unsafe`). Para Vale, embora a geração seja funcional, evidenciou-se que seu modelo de *single ownership* impõe um custo de desempenho substancial à estratégia de reutilização de estruturas do BenchGen, documentado aqui como uma limitação e ponto de partida para investigação futura.

Como trabalhos futuros, destacam-se: (i) a investigação do custo de alocação observado em D, que o mantém entre as linguagens compiladas mais custosas mesmo sob compilação LLVM (`ldc2`); (ii) a atualização do *backend* de Zig para a API da versão 0.16 e a evolução do *backend* nativo de V, hoje ainda muito mais lento que o *backend* C; (iii) a consolidação do suporte a Vale e sua eventual incorporação ao repositório oficial; e (iv) a ampliação do intervalo de profundidades medido em *hardware* mais robusto e termicamente estável, permitindo uma comparação de escalabilidade mais detalhada e menos sujeita a ruído.

Referências

[1] Silva, V. F., & Pereira, F. M. Q. (2024). Benchmark Generation via L-Systems. Repositório Oficial do BenchGen. Disponível em: https://github.com/lac-dcc/BenchGen.

[2] Lindenmayer, A. (1968). Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280-299.

[3] The V Programming Language. Disponível em: https://vlang.io.
[4] The Zig Programming Language. Disponível em: https://ziglang.org.
[5] The Odin Programming Language. Disponível em: https://odin-lang.org.
[6] The D Programming Language. Disponível em: https://dlang.org.
[7] The Nim Programming Language. Disponível em: https://nim-lang.org.
[8] Ada Programming Language. Disponível em: https://www.adacore.com/about-ada.
[9] The Vale Programming Language. Disponível em: https://vale.dev.