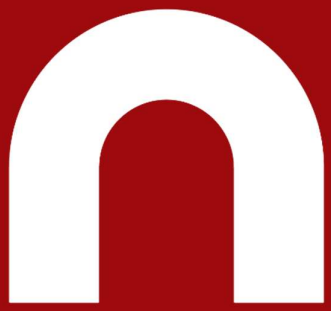


isec
Engenharia

MESTRADO EM ENGENHARIA INFORMÁTICA

DEFINITIVO

Linguagem Orientada a Aspetos para Transformação de Webassembly

Autor

João Paulo Mendes Rodrigues

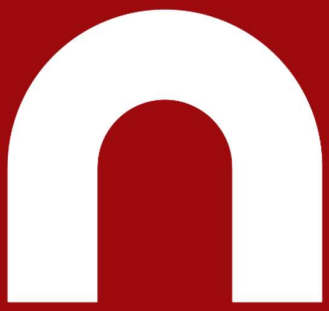
Orientador

Jorge Miguel Sousa Barreiros

INSTITUTO POLITÉCNICO DE COIMBRA

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

Coimbra, abril 2022



isec
Engenharia

DEPARTAMENTO DE INFORMÁTICA E SISTEMAS

Linguagem Orientada a Aspetos para Transformação de Webassembly

Relatório de Trabalho de Projeto para a obtenção do grau de Mestre em Engenharia Informática

Especialização em Desenvolvimento de Software

Autor

João Paulo Mendes Rodrigues

Orientador

Jorge Miguel Sousa Barreiros

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, abril 2022

AGRADECIMENTOS

Aos meus pais, irmã e avó, pelo apoio que me deram em todas as fases da minha vida, que se revelou crucial para a elaboração deste documento. Agradeço-lhes a sua compreensão e a habitual motivação com que pude contar ao longo deste trabalho.

À minha namorada, pela força e apoio incondicional que sempre me transmitiu para elaborar o projeto e pela total compreensão da minha ausência, em muitos dos programas que tiveram lugar durante este período.

Agradecer também ao meu orientador, o Doutor Jorge Miguel Sousa Barreiros, por toda a disponibilidade, orientação e apoio fornecidos.

RESUMO

A execução de um programa de Webassembly numa máquina cliente implica o descarregamento do código do servidor. Isto significa que qualquer alteração que o cliente pretenda fazer no código deve ser feita diretamente no código compilado. Estas transformações podem ser motivadas por diversas razões, tais como: reparação imediata de erros/problemas encontrados em produção, neutralização de código potencialmente malicioso, melhorias de desempenho, etc.

Outras aplicações de transformações diretas de código binário Webassembly noutros contextos incluem a instrumentação do código, a geração de um conjunto de variações a partir do mesmo código binário, por exemplo, com variações otimizadas para diferentes arquiteturas (no cliente ou servidor) – sem recompilação do programa, uma vez que existe sempre a versão base (compilada a partir do código fonte), e um conjunto de versões relativas às variações - ou a monitorização do desempenho das atividades realizadas pelo programa.

Com isto, o trabalho desenvolvido neste documento consiste numa ferramenta para manipulação de código WebAssembly que permite a realização de pesquisas no código, e a sua substituição por um conjunto de instruções definidas pelo programador. A ferramenta tem o nome de WasmManipulator e seguirá uma abordagem orientada a aspetos para alcançar a flexibilidade e simplicidade pretendidas. Para além disso, dado que rotinas de Webassembly podem ser fortemente interdependentes do código JavaScript que as usa, a ferramenta possui determinadas funcionalidades que permitem tirar partido e explorar essa dependência. Isto inclui a definição de tipos adicionais no código WASM, e interpretação/execução de expressões em tempo de execução.

O WasmManipulator foi desenvolvido com recurso à linguagem Go, e recorreu a outras tecnologias e ferramentas para se auxiliar no seu desenvolvimento, tais como, YAML, WABT e Comby. Para implementar o módulo JavaScript utilizou-se o TypeScript, que juntamente com o GulpJS, geraram o respetivo código.

Com a implementação de todos os requisitos estabelecidos para a ferramenta, validados através da utilização de cenários de uso típicos em código disponível publicamente, dão-se por atingidos os objetivos previstos para a ferramenta.

Palavras-Chave: WebAssembly, JavaScript, Aspetos, Aplicações Web, Transformação Binária, Instrumentação

ABSTRACT

Executing a Webassembly program on a client machine implies the download of the code from the server. This means that any changes that the client intends to make on the code must be made directly on the compiled version. These transformations can be motivated by several reasons, such as: immediate repair of errors/problems encountered in production, neutralization of potentially malicious code, performance improvements, etc.

Other applications of direct transformations of binary code Webassembly in another contexts include code instrumentation, the generation of a set of variations from the same binary code, for example, with variations optimized for different architectures (on the client or server) – without recompiling the program, since there is always the base version (compiled from the source code), and a set of versions relating to variations – or monitoring the performance of the activities carried out by the program.

With this, the work developed in this document consists of a tool for manipulating WebAssembly code that allows you to perform searches on the code, and its replacement by a set of instructions defined by the programmer. The tool is named WasmManipulator and follows an aspect-oriented approach to achieve the desired flexibility and simplicity. In addition, because Webassembly routines can be heavily interdependent on the JavaScript code that uses them, the tool has certain features that allow you to take advantage of and exploit this dependency. This includes defining additional types in WASM code, and interpreting/executing expressions at runtime.

WasmManipulator was developed using the Go language, and also used other technologies and tools to assist in its development, such as YAML, WABT and Comby. To implement the JavaScript module, TypeScript and GulpJS were used to generate the output code.

With the implementation of all the requirements established for the tool, which were validated by implementing typical usage scenarios on publicly available code, it is possible to state that the objectives defined for the tool have been achieved.

Keywords: WebAssembly, JavaScript, Aspects, Web Applications, Binary Transformation, Instrumentation

ÍNDICE

Capítulo 1. Introdução 1
1.1. Metodologia de Trabalho 2
1.2. Enquadramento Institucional 3
1.3. Estrutura do documento 3
Capítulo 2. Webassembly 5
2.1. Objetivos 5
2.1.1. Rapidez e eficiência 6
2.1.2. Segurança 6
2.1.3. Portabilidade 7
2.2. Conceitos 8
2.3. Estrutura 11
2.3.1. Módulo 11
2.3.2. Instruções 16
2.4. Máquina Virtual do WebAssembly 19
2.5. Interação com o JS 20
Capítulo 3. Trabalho relacionado 25
3.1. Tecnologias Existentes 25
3.1.1. Wasabi 25
3.1.2. WasmProf 26
3.1.3. WABT 26
3.1.4. WAIL 27
3.1.5. Análise Comparativa 28
3.2. Linguagens de Transformação 29
3.2.1. ASM 29
3.2.2. Javassist 29
3.2.3. AspectJ 30
3.2.4. Análise Comparativa 32
3.3. Linguagens para Procura e Substituição 33
3.3.1. Xpath e XSLT 33
3.3.2. CRAQL 34
3.3.3. Stratego/XT 34

3.3.4. Comby 35
3.3.5. Análise Comparativa 36
3.4. YAML 37
Capítulo 4. Especificação da Linguagem 39
4.1. Requisitos 39
4.2. Linguagem para transformação 40
4.2.1. Exemplo Inicial 40
4.2.2. Modo Inteligente 43
4.2.3. Adicionar contexto à instrumentação 45
4.2.4. Reverter Argumentos das Chamadas a Funções 46
4.2.5. Ordenar Aplicação das Transformações 48
4.2.6. Adicionar Multiplicação 48
4.2.7. Instrumentar Operações 50
4.2.8. Restringir Operações Instrumentadas 52
4.2.9. Contagem da Execução de Operações 53
4.2.10. Instrumentação com Informação Textual 54
4.2.11. Histórico da Execução de Operações 56
4.2.12. *Caching* das Operações 59
4.2.13. Execução do Exemplo 60
4.3. Execução da Ferramenta 62
4.4. Integração com o JS 63
4.5. Conclusão 64
Capítulo 5. Arquitetura e Fluxo de Desenvolvimento 65
5.1. Arquitetura de *packages* 66
5.2. Configuração da Ferramenta 68
5.2.1. Configuração da Execução 68
5.2.2. Configuração das Variáveis de Ambiente 69
5.2.3. Configuração do Serviço de *Logging* 70
5.2.4. Carregamento e Configuração dos *Go Templates* 70
5.3. Leitura dos Ficheiros de Entrada 71
5.4. Transformação do Módulo 71
5.4.1. Preenchimento dos Metadados 72

5.4.2. Execução dos *Pointcuts* 74
5.4.3. Transformações Globais 80
5.4.4. Transformação dos *Join-Points* 82
5.4.5. Transformação do Código das Novas Funções 83
5.4.6. Transformações *Runtime* 83
5.5. Geração de Resultados 90
Capítulo 6. Validação 91
6.1. Stooge Sorting 93
6.1.1. Transformação – *Logging* básico 94
6.1.2. Transformação – Monitorização de Desempenho 96
6.1.3. Transformação – Alterar para Algoritmo mais Rápido 97
6.1.4. Transformação – Planos de Subscrição 99
6.2. Filtros de Imagens 101
6.2.1. Transformação – Filtro Baseado na Função Seno 102
6.2.2. Transformação – Monitorização de Desempenho 106
6.2.3. Transformação – *Caching* 108
6.2.4. Transformação - Reparação de um Erro 110
6.2.5. Transformação - Filtro *Sepia* 111
6.2.6. Transformação – Melhoramentos com JS 120
6.3. Markdown *Parser* 125
6.4. Jogo Invaders 136
6.5. Discussão da Validação 139
Capítulo 7. Conclusão 141
7.1. Resumo do Trabalho 141
7.2. Contribuições 143
7.3. Trabalho Futuro 143
Referências Bibliográficas 145
Anexos 151
Anexo A. Opções de Configuração 153
Anexo B. Especificação Detalhada da Linguagem 157
1. Sintaxe 158
1.1. String 159

1.2. Identifier 159
1.3. Type 159
1.4. Variable 160
1.5. Code 160
1.6. Pointcut 161
1.7. Template 162
2. Pointcut Expressions 162
2.1. Pointcut func 163
2.2. Pointcut call 166
2.3. Pointcut args 167
2.4. *Pointcut* returns 167
2.5. *Pointcut* template 168
3. Code Expressions 169
3.1. Static Expressions 169
3.2. Runtime Expressions 177
3.3. Template Expressions 179
4. Modo Inteligente (Smart) 181

ÍNDICE DE FIGURAS

Figura 1 - Estrutura de funcionamento da ferramenta WasmManipulator 2
Figura 2 - Código C/C++ para conversão em WASM 7
Figura 3 - Código WAT para conversão em WASM 7
Figura 4 - Código Rust para conversão em WASM 8
Figura 5 - Código AssemblyScript para conversão em WASM 8
Figura 6 – Acesso e manipulação da memória linear WASM 9
Figura 7 – Código WAT que inclui o lançamento de uma *trap* 9
Figura 8 – Código WAT e JS com a inicialização do elemento *table* (MDN Contributors, Understanding WebAssembly text format, 2021) 10
Figura 9 – Fases da semântica do WASM (Atapattu, 2020) 11
Figura 10 – Código WAT com um exemplo da definição do elemento tipo 11
Figura 11 – Código WAT com um exemplo da definição do elemento função 12
Figura 12 – Código WAT com um exemplo da definição dos elementos tabela e segmentos de dados 12
Figura 13 – Código WAT com um exemplo da definição dos elementos memória e segmentos de dados 13
Figura 14 – Código WAT com um exemplo de um módulo WASM 13
Figura 15 - Módulo WASM para gerar série Fibonacci artigo (Sendilkumarn, 2020) 16
Figura 16 - Código WAT com exemplos de instruções numéricas 17
Figura 17 - Código WAT com exemplos de instruções paramétricas 17
Figura 18 - Código WAT com exemplos de instruções para acesso a variáveis 17
Figura 19 - Código WAT com exemplos de instruções de acesso à memória 18
Figura 20 - Código WAT com exemplos de instruções de gestãoo de memória 18
Figura 21 - Código WAT com exemplos de expressões 18
Figura 22 - Código WAT com exemplos de instruções de controlo 18
Figura 23 - Código JS com a inicialização do módulo WASM (ICHI.PRO, 2021) 21
Figura 24 - Código JS com o carregamento do módulo WASM através do método *instantiateStreaming* 21
Figura 25 - Código JS com importação do objeto no módulo WASM (ICHI.PRO, 2021). 21
Figura 26 - Código WASM com a definição da função importada *log* (ICHI.PRO, 2021) 22
Figura 27 - Código WASM com a definição da função *printNumber* exportada (ICHI.PRO, 2021) 22

Figura 28 - Código JS com a chamada à função exportada *printNumber* do módulo (ICHI.PRO, 2021).....22
Figura 29 - Código AspectJ com AOP aplicado a uma aplicação Java (AspectJ, 2001) 31
Figura 30 - Resultado da execução das transformações realizada a uma aplicação Java com o AspectJ (AspectJ, 2001) 32
Figura 31 - Código Go com exemplo de entrada do Comby (Comby, 2021) 35
Figura 32 - Código JSON com o resultado do exemplo do Comby (Comby, 2021).....35
Figura 33 - Código de entrada para procurar por padrão no Comby (Comby, 2021) 36
Figura 34 - Código YAML com exemplo da linguagem (Erik, 2018).....38
Figura 35 - Modelo de dados conceptual associado ao exemplo da linguagem YAML 38
Figura 36 - Código WAT inicial para o exemplo da especificação da linguagem.....41
Figura 37 – Código com a transformação para instrumentar exemplo da especificação da linguagem 41
Figura 38 - Código WAT inicial para o exemplo da especificação da linguagem com o *join-point* assinalado 42
Figura 39 - Código WAT resultante da transformação para instrumentar exemplo da especificação da linguagem.....43
Figura 40 - Código WAT atualizado com a inserção da operação de subtração no exemplo da especificação da linguagem 44
Figura 41 - Código com a transformação inteligente para corrigir a instrumentação adicionada ao exemplo da especificação da linguagem 44
Figura 42 – Código WAT resultante da transformação inteligente para instrumentar exemplo da especificação da linguagem 45
Figura 43 – Código com a transformação para inserir quais as funções que participam na instrumentação adicionada ao exemplo da especificação da linguagem.....46
Figura 44 - Código com a transformação para reverter a ordem dos operadores no exemplo da especificação da linguagem 47
Figura 45 - Código com a transformação ordenada que corrige a transformação que reverte a ordem dos operadores no exemplo da especificação da linguagem 48
Figura 46 - Código com a transformação para inserir a operação de multiplicação no exemplo da especificação da linguagem 49
Figura 47 - Código com a transformação onde é feito o acesso ao contexto do *pointuct* inserir a operação de multiplicação no exemplo da especificação da linguagem.....50

Figura 48 - Código com a transformação onde é feito o registo das operações executadas no exemplo da especificação da linguagem 51
Figura 49 - Código com a transformação onde é feito a combinação de *templates* para proteger as operações a que o registo das operações é feito no exemplo da especificação da linguagem 53
Figura 50 - Código com a transformação para realizar a contagem do número de vezes que uma operação é executada no exemplo da especificação da linguagem 54
Figura 51 - Código com a transformação onde são aplicadas *strings* nas transformações de instrumentação aplicadas ao exemplo da especificação da linguagem 56
Figura 52 - Código com a transformação onde é guardado o histórico da execução das operações no exemplo da especificação da linguagem 58
Figura 53 - Código com a transformação onde é implementada uma *cache* nas operações do exemplo da especificação da linguagem 60
Figura 54 - Código JS que executa as transformações realizadas no código do exemplo da especificação da linguagem 61
Figura 55 - Resultado na consola da execução do código da especificação da linguagem transformado 62
Figura 56 - Exemplo do comando para executar a ferramenta WasmManipulator 62
Figura 57 - Código JS com integração do módulo JS produzido pela ferramenta com transformações *runtime* 64
Figura 58 - Fluxo interno com os passos da execução da ferramenta 65
Figura 59 - Esquema arquitetural dos packages na ferramenta 66
Figura 60 - Estrutura da diretoria com as dependências da ferramenta ("dependencies/") 69
Figura 61 - Resultado do *parser* interno após análise de código de transformação 73
Figura 62 - Árvore sintática resultante do *parse* de uma expressão de *pointcut* 76
Figura 63 - Código WAT para exemplo com a função *join* 86
Figura 64 - Contexto incial do JS para o exemplo com a função *join* 86
Figura 65 - Contexto do JS no 1º passo do exemplo com a função *join* 86
Figura 66 - Contexto do JS no 2º passo do exemplo com a função *join* 87
Figura 67 - Contexto do JS no 3º passo do exemplo com a função *join* 87
Figura 68 - Contexto do JS no 4º passo do exemplo com a função *join* 87
Figura 69 - Contexto do JS no 5º passo do exemplo com a função *join* 87
Figura 70 - Contexto do JS no 6º passo do exemplo com a função *join* 87
Figura 71 - Contexto do JS no 7º passo do exemplo com a função *join* 87

Figura 72 - Contexto do JS no 8º passo do exemplo com a função *join* 87
Figura 73 - Contexto do JS no 9º passo do exemplo com a função *join* 87
Figura 74 - Contexto do JS no 10º passo do exemplo com a função *join* 87
Figura 75 - Contexto do JS no 11º passo do exemplo com a função *join* 87
Figura 76 - Contexto do JS no 12º passo do exemplo com a função *join* 88
Figura 77 - Contexto do JS no 13º passo do exemplo com a função *join* 88
Figura 78 - Contexto do JS no 14º passo do exemplo com a função *join* 88
Figura 79 - Contexto do JS no 15º passo do exemplo com a função *join* 88
Figura 80 - Contexto do JS no 16º passo do exemplo com a função *join* 88
Figura 81 - Contexto do JS no 17º passo do exemplo com a função *join* 88
Figura 82 - Contexto do JS no 18º passo do exemplo com a função *join* 88
Figura 83 - Contexto do JS no 19º passo do exemplo com a função *join* 88
Figura 84 - Código da transformação para o *logging* básico da função de ordenação 95
Figura 85 - Código JS para o *logging* básico da função de ordenação 96
Figura 86 - Resultado na consola da transformação para o *logging* básico da função de ordenação 96
Figura 87 - Código da transformação para a monitorização de desempenho da função de ordenação 96
Figura 88 - Resultado na consola da transformação para a monitorização de desempenho da função de ordenação 97
Figura 89 - Código da transformação para um algoritmo de ordenação mais rápido 99
Figura 90 - Resultado na consola da transformação para um algoritmo de ordenação mais rápido 99
Figura 91 - Código da transformação com o plano de subscrição para a função de ordenação 100
Figura 92 - Código JS para a transformação com o plano de subscrição para a função de ordenação 101
Figura 93 - Resultado na consola da execução da função de ordenação sem o melhor plano 101
Figura 94 - Resultado na consola da execução da função de ordenação com o melhor plano 101
Figura 95 - Imagens de teste na aplicação de filtros de imagens ("picture1.jpg" e "picture2.jpg") 102
Figura 96 - Código da transformação que adiciona o filtro “sin” 104

Figura 97 - Código JS para a execução do filtro “sin” após transformação GitHub (Tulka, 2021) 106
Figura 98 - Resultado da execução do filtro “sin” sobre as imagens de teste "picture1.jpg" e "picture2.jpg” 106
Figura 99 - Código da transformação para a monitorização de desempenho da função para o filtro “sin” 107
Figura 100 - Código JS com a monitorização de desempenho para a aplicação dos filtros 107
Figura 101 - Resultado na consola da monitorização de desempenho da aplicação do filtro “sin” nas imagens de teste 108
Figura 102 - Código da transformação de *caching* para o filtro "sin" 109
Figura 103 - Resultado na consola da aplicação do filtro "sin" com *cache* para as duas imagens de teste 109
Figura 104 - Código da transformação que corrige o erro na expressão do filtro "sin" 111
Figura 105 - Código com a implementação das funções "sepia" e "sepia_100000_times" 113
Figura 106 - Código com a atualização da transformação para a monitorização de desempenho das funções de filtro 115
Figura 107 - Código JS com a nova função de monitorização de desempenho das funções de filtro 115
Figura 108 - Resultado na consola após execução da função que aplica cem mil vezes o filtro *sepia* 115
Figura 109 - Código da transformação com a substituição das chamadas pelo conteúdo da função "sepia" 116
Figura 110 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* com a remoção das chamadas ao “sepia” implementado 117
Figura 111 - Código da transformação que aplica a otimização *Loop Unrolling* à função que aplica cem mil vezes o filtro *sepia* 118
Figura 112 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* após a otimização *Loop Unrolling* realizada para a mesma 119
Figura 113 - Código da transformação que aplica a otimização *Loop Unrolling* à função *sepia* 120
Figura 114 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* após a otimização *Loop Unrolling* realizada na função *sepia* 120

Figura 115 - Código da transformação que permite a monitorização partilhado pela mesma função 122
Figura 116 - Código JS com as alterações para implementar a monitorização partilhado 123
Figura 117 - Código da transformação para inclusão de uma *cache* para múltiplas funções 125
Figura 118 - Código da transformação para análise do programa *markdown-wasm* 125
Figura 119 - Código da transformação com a contagem de elementos no programa *markdown-wasm* 132
Figura 120 - Código JS modificado para suportar a contagem de elementos Markdown. 135
Figura 121 - Resultado da execução do programa *markdown-wasm* com a funcionalidade de contagem 136
Figura 122 - Código da transformação com os novos modos de jogo no Invaders 138
Figura 123 - Código JS modificado para suportar os novos modos de jogo no Invaders . 139
Figura 124 - Estrutura YAML da linguagem de transformação do WasmManipulator 157
Figura 125 – Estrutura YAML linguagem de transformação do WasmManipulator com os tipos especificados 159
Figura 126 - Código com limitação das *runtime references* 178
Figura 127 - Código WAT original para o modo "inteligente" 181
Figura 128 - *Pointcut expression* para transformação no modo "inteligente" 181
Figura 129 - Código do *advice* para a transformação no modo "inteligente" 181
Figura 130 - Código WAT resultante sem o modo "inteligente" ativo 181
Figura 131 - Código WAT resultante com o modo "inteligente" ativo 182

ÍNDICE DE TABELAS

Tabela 1 – Propriedades dos componentes do módulo WASM 14
Tabela 2 – Análise comparativa das tecnologias existentes 28
Tabela 3 - Análise comparativa das ferramentas de transformação para tecnologias análogas as WasmManipulator 32
Tabela 4 - Análise comparativa das ferramentas de pesquisa e substituição 36
Tabela 5 - Configurações da ferramenta WasmManipulator 63
Tabela 6 - Fluxo de obtenção dos join-points 76
Tabela 7 - Resumo de funcionalidades e técnicas aplicadas nas transformações de cada exemplo 91
Tabela 8 - Resultados da análise realizada ao fluxo do programa *markdown-wasm* 126
Tabela 9 –Elementos presentes na transformação do programa *markdown-wasm* 133
Tabela 10 - Inicialização do valor das variáveis na ferramenta WasmManipulator 160
Tabela 11 - Elementos da função de *pointcut* func 163
Tabela 12 – Modelo de dados de contexto da função de *pointcut* func 165
Tabela 13 – Modelo de dados de contexto da função de *pointcut* call 166
Tabela 14 - Modelo de dados para os dados de contexto do tipo “argumento" 167
Tabela 15 - Modelo de dados de contexto da função de *pointcut* returns 168
Tabela 16 - Funções de transformação para *static expressions* 170
Tabela 17 - Funções para combinação de *templates* nas *template keywords* 180

CAPÍTULO 1.
INTRODUÇÃO

Tradicionalmente as aplicações *web* são desenvolvidas com linguagens como JavaScript, C# e Java. O JavaScript (JS) é uma linguagem utilizada nas aplicações do lado do cliente, que por ser interpretada não precisa de ser compilada. Apesar de possuir diversas características positivas para o desenvolvimento de aplicações *web*, como simplicidade, interoperabilidade, eficiência e versatilidade, não é ideal para todo o tipo de aplicações (Team D. , 2019), tais como, edição de vídeo, renderização 3D, videojogos, e encriptação, pois não está concebido para executar aplicações que exigem um processamento intensivo (Optasy, 2018), estando longe de executar com o mesmo desempenho que o sistema operativo para um programa comparável mas nativo (Bryant, 2017).

Para responder a este desafio, a equipa do Mozilla, juntamente com outros fabricantes de navegadores, criaram uma nova linguagem padrão, o WebAssembly (WASM), que pode ser executado como parte integrada dos navegadores *web* (Bryant, 2017). O WASM é uma linguagem que utiliza/define um conjunto de instruções binárias e que foi criada para obter melhor desempenho face a aplicações baseadas apenas em JS ou outras tecnologias *web*, tanto no lado do cliente como no lado do servidor (WebAssembly, 2017). Esta tem vindo a crescer à medida que muitos escolhem a linguagem para a implementação de aplicações que exijam elevado processamento na *web*, sendo algo que anteriormente não era possível.

O código WASM é previamente compilado e enviado pelo servidor para o cliente. O código executado pelo cliente encontra-se num formato binário, e por isso, torna-se difícil para o cliente controlar e alterar esse código. No entanto, a capacidade de alterar o código do lado do cliente, sem implicar alteração direta do código fonte, pode ser útil em diversos cenários (Restivo, 2006), tais como:

- Instrumentação.
- Monitorização de desempenho.
- Segurança e autenticação.
- Melhorias de desempenho.
- Reparação de erros (*hotfixes*).
- *Caching*.
- Estender/alterar funcionalidades.

Quando são considerados outras linguagens/ecossistemas, tais como Java (Team O. , Java, 2021) ou C/C++ (Team C. , 2021), existem ferramentas específicas para aplicar este tipo de modificações (sobre o formato binário do programa) (Team A. , 2021) (JBoss-Javassist, 2020) (AspectJ, 2001) (Spinczyk & GmbH, 2021). No entanto, as soluções disponíveis para WASM são poucas e muito limitadas (Lab, 2020) (Gifford, 2019) (Smith & Community, 2021) (Qwokka, 2019), em termos das transformações que permitem realizar. Isto pode dever-se não só ao facto de a tecnologia ser relativmente recente, mas também ao facto do WASM possuir um funcionamento articulado com JS, o que dificulta a implementação de

modificações sem impactar também a componente de JS. Este trabalho visa então resolver esse problema, tendo como principal objetivo a criação de uma ferramenta, o WasmManipulator (Rodrigues, 2021), que deverá permitir a realização das alterações para cenários deste tipo.

Na Figura 1 encontra-se ilustrado um esquema do funcionamento geral da ferramenta, onde um módulo WASM e um ficheiro com as transformações é introduzido na ferramenta WasmManipulator, e após a aplicação das transformações é gerado o respetivo resultado. Dependendo do tipo de transformações aplicado, os ficheiros gerados podem ser diferentes. Isto é, o módulo WASM transformado é sempre gerado, contudo, pode ser necessário gerar um ficheiro com um módulo JS auxiliar para o caso onde as transformações possuam alguma definição *runtime*. Desta forma, se for gerado o módulo JS auxiliar, o cliente deverá interagir com este ao invés do módulo WASM.

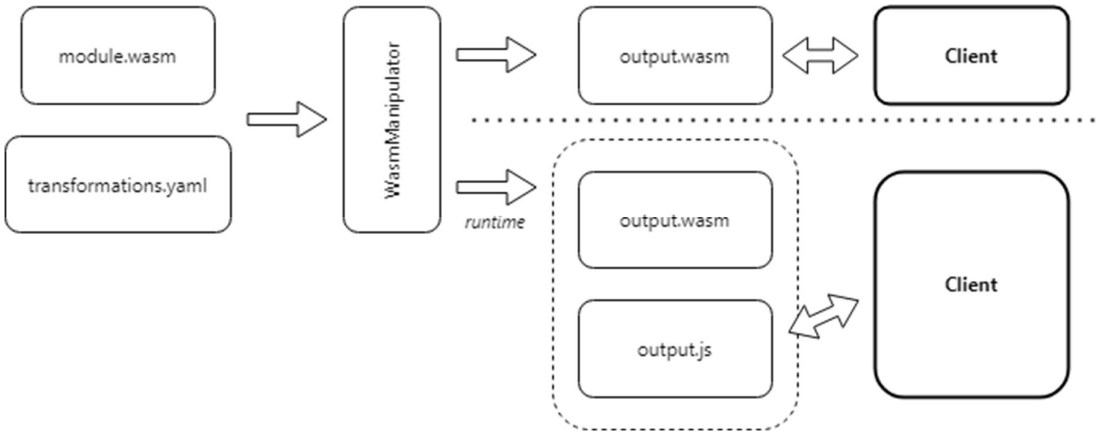


Figura 1 - Estrutura de funcionamento da ferramenta WasmManipulator

Para atingir a flexibilidade desejada foi adaptado um paradigma orientado a aspetos. Isto acontece porque este paradigma promove o desacoplamento entre as alterações e o código original, facilitando assim a manutenção e legibilidade destas alterações, e reduzindo a desordem e complexidade do código . Para além disto, permite a reutilização do código das alterações, uma vez que cada definição pode ser aplicada em vários pontos do código (Kanjilal, 2016).

1.1. Metodologia de Trabalho

Relativamente à metodologia utilizada no presente documento, de forma a desenvolver este trabalho, foi realizada uma identificação do problema e de soluções disponíveis, conceptualização da solução incluindo requisitos/funcionalidades e casos de uso, procura e análise de tecnologias de suporte, desenvolvimento e validação. De forma a validar o trabalho foi realizada uma prova de conceito baseada em exemplos, onde foram identificados um conjunto de aplicação do contexto real que permitiram demonstrar a utilidade e validade da ferramenta.

O trabalho vem sendo feito ao longo de 8 meses, onde durante o processo foi feita uma apresentação intermédia e reuniões semanais por videoconferência. Concluído este trabalho,

será submetido para publicação um artigo sobre a ferramenta alvo deste documento, para a conferência CISTI'2022 - 17ª Conferência Ibérica de Sistemas e Tecnologias de Informação.

1.2. Enquadramento Institucional

Este trabalho foi desenvolvido e apresentado no âmbito do documento "Linguagem Orientada a Aspetos para Transformação de Webassembly" que se enquadra no Mestrado em Engenharia Informática, do Departamento de Informática e Sistemas (DEIS), no ramo de Desenvolvimento de Software, do Instituto Superior de Engenharia de Coimbra (ISEC).

O ISEC (ISEC, 2021) é uma instituição de ensino superior orientada para a formação de nível superior focada no exercício de atividades profissionais no domínio da Engenharia, com uma rica e vasta história criada ao longo dos seus noventa anos de existência.

1.3. Estrutura do documento

Neste documento será apresentado o design, implementação e análise da ferramenta WasmManipulator, que consiste numa ferramenta de transformação de código WASM[1]. Por ser uma tecnologia relativamente recente, o ecossistema do WASM encontra-se em constante expansão. WasmManipulator será uma das muitas ferramentas que contribuem para o crescimento deste ecossistema, ajudando a que esta linguagem evolua cada vez mais.

O restante deste documento está organizado da seguinte forma:

- Capítulo 2 – Apresenta uma breve introdução ao WASM, onde é feita uma abordagem sobre o estado atual da tecnologia, que inclui alguns dos seus objetivos, conceitos e estrutura. É também feita uma breve explicação sobre a integração de WASM num programa JS.
- Capítulo 3 – Aborda uma análise das ferramentas e linguagens para WASM com objetivo semelhante ao do WasmManipulator. Para além disso, aborda a análise de ferramentas similares, mas para outras linguagens ou ecossistemas. Também se encontra descrita uma análise de um conjunto de ferramentas de pesquisa e substituição que foram consideradas para a funcionalidade de pesquisa por padrão, necessária para implementação de alguns aspetos da linguagem descrita neste trabalho. Por último, será feita uma curta introdução sobre o YAML, que é a linguagem utilizada como base para derivar a linguagem de transformação descrita neste trabalho.
- Capítulo 4 – Apresenta a especificação da ferramenta, incluindo os requisitos, uma introdução à linguagem de transformação através de exemplos, a configuração necessária para a ferramenta ser executada, e um guia para integração com uma aplicação JS.
- Capítulo 5 – Contém a arquitetura e o fluxo de execução presentes na implementação da ferramenta.

[1] A ferramenta na verdade transforma código WASM e JS. No entanto, por conveniência e brevidade, limitamo-nos a referir no texto apenas a vertente de manipulação de WASM aquando da designação da ferramenta.

- Capítulo 6 – Este capítulo apresenta a validação do trabalho. Para este efeito, foi realizada uma prova de conceito baseada em exemplos, onde foi definido um conjunto de aplicações da ferramenta baseados em casos de contexto real. As implementações apresentadas neste capítulo usam maioritariamente o JS como o ambiente hospedeiro para instanciar os programas WASM.
- Capítulo 7 – Este capítulo conclui o documento. É apresentado um resumo do trabalho realizado, são identificadas as contribuições e elencadas propostas de trabalho futuro.

CAPÍTULO 2.
WEBASSEMBLY

Para melhor compreender o trabalho, é útil conhecer a tecnologia que será o alvo da ferramenta desenvolvida, o WASM. Este consiste numa linguagem de instrução binária criada para obter melhor desempenho nas aplicações *web*, tanto no lado do cliente como no lado do servidor (WebAssembly, 2017). Apesar do nome, WASM não é estritamente uma linguagem *Assembly* uma vez que não se destina a nenhuma máquina em específico, mas sim a uma máquina virtual baseada em pilha. Isto significa que pode ser executada numa ampla variedade de plataformas, incluindo o navegador *web*, que após o carregamento é imediatamente capaz de transformá-lo para qualquer formato *Assembly*. Para além de ser conhecido por ser rápido e eficiente, é também seguro, uma vez que cria um ambiente isolado e controlado, aplicando políticas de segurança e as respetivas permissões do navegador (Mihajlija, 2019).

A Secção 2.1 irá abordar os principais objetivos estabelecidos para a linguagem WASM, que inclui a rapidez e eficiência, segurança e portabilidade.

Na Secção 2.2 serão abordados os conceitos presentes na linguagem, que engloba a forma como a linguagem opera dentro do seu modelo de computação baseado em pilha, a sua disposição através de módulos WASM, os tipos de valores e os principais elementos disponibilizados, e por fim, as fases presentes na sua semântica.

O código WASM é sempre disposto num módulo WASM que segue uma determinada estrutura para que os vários elementos (tais como, funções, memórias lineares, etc.) sejam devidamente definidos. A Secção 2.3 possui a informação relativa a esta estrutura, onde vão ser detalhados cada um destes elementos (secções do módulo), e ainda as instruções que compõem o código das funções e permitem a manipulação dos valores da pilha.

O WASM é executado numa máquina virtual já instalada nos navegadores *web*, sendo esta compartilhada com o JS. Com isto, na Secção 2.4 será descrita uma curta análise sobre o estado atual desta máquina virtual, onde serão apresentados os motivos pelo qual foi optado por criar um novo formato binário ao invés de ser utilizado um dos vários formatos que já existem para o mesmo fim.

Por fim, a Secção 2.5 vai apresentar o JS como uma linguagem essencial para integrar com o WASM de forma a tirar partido da melhoria de desempenho oferecida pelo mesmo, para que sejam desenvolvidas aplicações *web* que antes ou não eram possíveis ou eram extremamente lentas devido ao elevado processamento requerido.

2.1. Objetivos

De acordo com a especificação da linguagem (Rossberg, 2021), esta tem como objetivos principais a rapidez e eficiência, segurança e portabilidade. Além disso, é desenhada para que o código WASM também seja fácil de analisar e inspecionar, especialmente em ambientes como navegadores da *web*. Nesta secção serão detalhados cada um destes objetivos, assim como possíveis problemas que possam existir nos mesmos.

2.1.1. Rapidez e eficiência

O WASM pretende executar código com um desempenho próximo de código nativo, aproveitando ao máximo os vários recursos comuns a todos os *hardwares* contemporâneos. Para isso, o código ao ser importado na máquina do cliente deve ser compilado para *Assembly*, e a máquina virtual que o executa é responsável por aplicar as otimizações necessárias de acordo com a arquitetura do respetivo *hardware* que o executa (conforme descrito nas Secções 2.1.2 e 2.1.3).

O código WASM pode ser decodificado, validado e compilado numa única passagem rápida, paralela à compilação, Just-in-Time (JIT) ou Ahead-of-Time (AOT). Com isto, e juntando o facto do WASM ter sido concebido para permitir a compilação em *streaming*, onde é feita a divisão em múltiplas tarefas paralelas independentes, faz com que o tempo de compilação de um programa seja mínimo.

Para além disso, o WASM é extremamente rápido quando comparado com tecnologias mais recentes que possuem objetivos semelhantes, tais como, o AsmJS (Team A. , 2021), Java Applets (Team O. , Java Applets, 2021), etc. Segundo o artigo (Atapattu, 2020), WASM é capaz de superar as otimizações realizadas pelo AsmJS, diminuir o tempo de descarregamento do código uma vez que possui código de tamanho menor, e ainda não precisa de realizar nenhum *parse* inicial, pois já se encontra num formato binário. Adicionalmente, a linguagem foi projetada para utilizar recursos da CPU (Central Processing Unit) que não estão disponíveis no JS (por exemplo, os tipos inteiros de *64-bits*).

2.1.2. Segurança

Quando executado no navegador, os módulos WASM são geridos pela mesma máquina virtual que executa o código JS. Desta forma, os problemas de segurança que ocorrem são os mesmos que poderão ocorrer no código JS. No entanto, pelo facto de que o código executa significativamente mais rápido que o JS, os problemas poderão surgir mesmo antes de serem detetados pelo antivírus, uma vez que este fator dificulta a verificação dos programas em busca de instruções maliciosas. Apesar disso, existem diversas propriedades e restrições que tentam tornar o WASM o mais seguro possível, tais como, o facto de ser executado num ambiente controlado, isto é, uma *sandbox* controlada pela máquina virtual, ou o facto das aplicações WASM possuírem um acesso muito restrito ao espaço de memória que lhes é disponibilizado (Fioretti, 2021).

O código ao ser executado numa *sandbox* não só perde a visibilidade da máquina do cliente, como também é impedido de interagir diretamente com a mesma. Este acesso apenas ocorre através do WebAssembly System Interface (WASI), fornecido pela máquina virtual, que segue um modelo de segurança que garante que as aplicações só possam aceder a arquivos e diretorias aos quais tenham acesso. Para além disto, este ambiente assegura que a "pilha" de chamadas está inacessível ao programa, impossibilitando assim que aconteçam os tradicionais ataques de destruição de pilha sobre os endereços de retorno. Os ponteiros são também protegidos, uma vez que são compilados para índices que representam os respetivos deslocamentos na memória linear, e assim oculta detalhes da implementação, tais como os

endereços virtuais. A execução de um programa WASM possui sempre um comportamento conhecido pela máquina virtual, uma vez que este é definido para que seja fácil de interpretar, e assim prever se o programa é válido para execução ou não. Isto engloba que o destino para qualquer transferência de controlo seja devidamente conhecido, isto é, o destino para as instruções referentes a chamadas ou ramificações diretas e indiretas é sempre conhecido inicialmente, e assim, torna-se impossível chamar acidentalmente uma instrução errada no meio de uma função ou criar uma ramificação para fora de uma função (Wasmtime, 2021).

2.1.3. Portabilidade

A linguagem WASM foi desenhada com o intuito de ser independente não só do *hardware*, como foi mencionado acima, mas também da plataforma onde é executado. Desta forma, a linguagem pode ser compilada para qualquer arquitetura moderna, computador, dispositivos móveis ou sistemas embutidos. Assim, os programas podem operar em diversos ambientes de forma simples e universal, não fazendo qualquer suposição sobre a arquitetura onde são executados.

Atualmente existem diversas formas de gerar código WASM, tais como:

- converter código de outras linguagens, tais como, C/C++ (Team C. , 2021) (Figura 2);
- recorrer ao formato textual WASM (WAT) (Figura 3);
- compilar código de outras linguagens que utilizam uma sintaxe pré-definida que possibilita a conversão em WASM (Figura 4), tipo Rust (Team R. , 2021) ou Go (Team G. , 2021);
- ou utilizar AssemblyScript (MDN Contributors, WebAssembly Concepts, 2021) (Figura 5).

```c
int add(int a, int b) {
  return a + b;
}
```

Figura 2 - Código C/C++ para conversão em WASM

```
(module
 (export "add" (func $add))
 (func $add (param $0 i32) (param $1 i32) (result i32)
  (i32.add
   (local.get $0)
   (local.get $1)
  )
 )
)
```

Figura 3 - Código WAT para conversão em WASM

```rust
#[no_mangle]
pub extern "C" fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Figura 4 - Código Rust para conversão em WASM

```typescript
export function add(a: i32, b: i32): i32 {
    return a + b;
}
```

Figura 5 - Código AssemblyScript para conversão em WASM

Linguagens como C/C++ conseguem ser convertidas em código WASM sem necessitar de qualquer alteração no código fonte. Relativamente a outras linguagens como Rust ou Go, é necessário introduzir no código fonte instruções específicas que permitam a sua compilação para WASM. Em relação às linguagens específicas para WASM, a diferença é que o WAT é mais baixo nível que o AssemblyScript. AssemblyScript é uma alternativa baseada em TypeScript (Team T. , 2021) que facilita o desenvolvimento de WASM ao abstrair o programador de determinados detalhes, tais como, a interação/manipulação da "pilha" WASM, gestão dos índices dos elementos (WASM) presentes no programa, ...

Estas conversões para código WASM só são possíveis porque esta foi projetada para ser independente de qualquer linguagem em específico, não privilegiando nenhuma linguagem ou modelo de programação em específico.

2.2. Conceitos

O modelo de computação do WASM é baseado numa máquina em pilha. As instruções manipulam valores na pilha e podem ser divididas em duas categorias principais: instruções simples e de controlo. As instruções simples têm como objetivo executar operações que provocam uma alteração nos valores da pilha ("puxar" e "empurrar" valores na pilha), enquanto as instruções de controlo são responsáveis para alteração no fluxo de execução de um programa, como por exemplo, ciclos ou chamadas a funções.

O WASM é uma linguagem de programação semelhante a um *Assembly* de baixo nível que suporta apenas quatro tipos de valores, inteiros e reais (IEEE 754-20196) de 32 e 64 *bits*. Um dado programa encontra-se sempre contido num módulo WASM, que contém as definições para os vários elementos do programa, incluindo as funções e respetivos tipos, tabelas, memórias lineares e variáveis globais (ver Secção 2.3.1). Cada uma destas definições, exceto os tipos das funções, podem ser importadas ou exportadas para o ambiente hospedeiro, sendo que para isso, é necessário atribuir-lhes um determinado nome. Para além disto, o módulo pode também inicializar os dados para a sua memória e/ou tabela através da cópias de segmentos com um determinado deslocamento, e ainda definir uma função inicial que será automaticamente executada no início do programa. Neste momento, um módulo só

pode conter uma instância de memória e tabela, contudo de acordo com a especificação, esta restrição será alterada no futuro.

A memória linear consiste num *array* contíguo e mutável de *bytes*. O acesso a esta memória é feito de forma síncrona, e pode ser realizado tanto pelo WASM como pelo ambiente hospedeiro (Figura 6). Com isto, a memória tanto pode ser instanciada no módulo como pode ser recebida do ambiente hospedeiro, sendo que a sua criação dependerá do ambiente onde está incluído (Sletten, 2021). A memória é sempre criada com um tamanho inicial, podendo ser aumentada dinamicamente durante a execução do programa. Um acesso indevido à memória gera sempre uma *trap*.

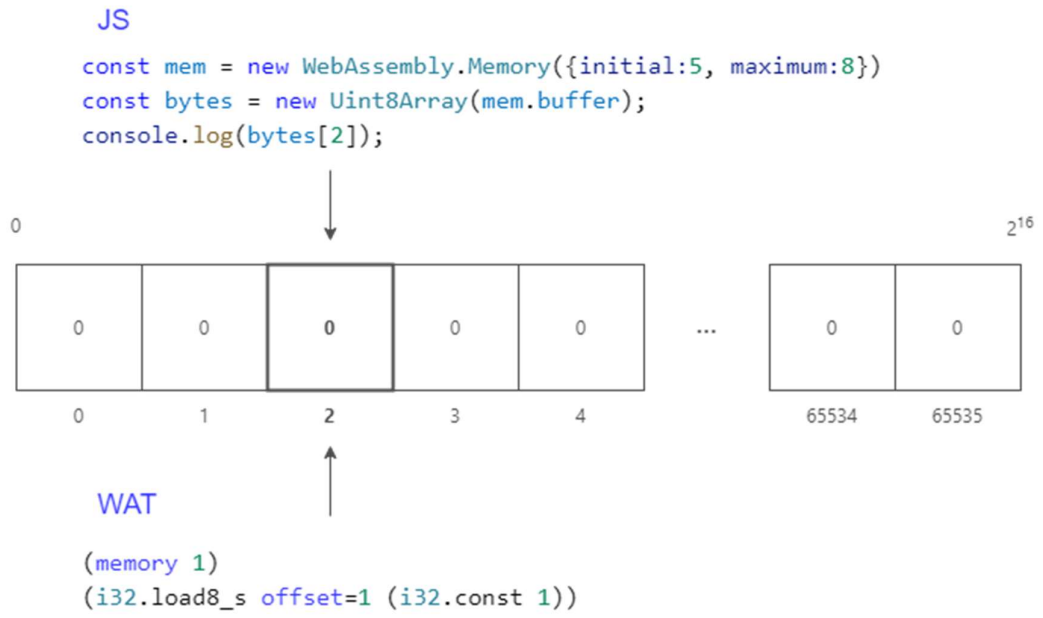


Figura 6 – Acesso e manipulação da memória linear WASM

As *traps* são elementos do WASM que representam erros lançados pelo módulo que abortam por completo a execução do programa. Estes erros não podem ser tratados internamente pelo módulo mas são sempre comunicados para o exterior. O código da função `$out_of_bounds` na Figura 7 iria provocar uma *trap*, uma vez que se está a aceder fora dos limites da memória linear (máximo $2^{16} - 1$).

```
(func $out_of_bounds (result i32)
  (i32.load8_s
    (i32.const 65536)) ;; [error]: RuntimeError: memory access out of bounds
)
(memory 1)
```

Figura 7 – Código WAT que inclui o lançamento de uma *trap*

Relativamente às tabelas, este elemento é responsável por permitir ao programa selecionar um determinado valor através do respetivo índice dinâmico. A sua estrutura consiste num *array* de valores opacos com um tipo específico. De acordo com a especificação, atualmente

só existem referências para elementos do tipo função. Na Figura 8, retirada de (MDN Contributors, Understanding WebAssembly text format, 2021), encontra-se ilustrada a inicialização da mesma tabela no módulo WASM (lado esquerdo) e no código JS (lado direito).

```
(module                              function() {
  (table 2 funcref)                    // table section
  (elem (i32.const 0) $f1 $f2)         var tbl = new WebAssembly.Table({initial:2
  (func $f1 (result i32)               , element:"funcref"});
    i32.const 42)
  (func $f2 (result i32)                 // function sections:
    i32.const 13)                        var f1 = ... /* some imported WebAssembly
  ...                                  function */
)                                        var f2 = ... /* some imported WebAssembly
                                       function */

                                         // elem section
                                         tbl.set(0, f1);
                                         tbl.set(1, f2);
                                       };
```

Figura 8 – Código WAT e JS com a inicialização do elemento *table* (MDN Contributors, Understanding WebAssembly text format, 2021)

Por último, existe uma entidade chamada *embedder* que é responsável por configurar o ambiente hospedeiro da execução do módulo. Este ambiente define como deve ser iniciado o carregamento do módulo, como serão disponibilizados os elementos importados e como será feito o acesso aos elementos exportados.

Conceptualmente, a semântica do WASM é dividida em três fases. A primeira consiste na fase de decodificação, onde é feita a formatação e conversão do módulo WASM para o respetivo código máquina. Depois, ocorre a fase de validação, em que é feita a verificação de uma série de condições que visam criar as condições necessárias para garantir que o módulo é válido e seguro. Em particular, esta fase executa validações de tipo para funções e respetivas instruções. Por fim, ocorre a fase de execução, que é formada pela instanciação do módulo e a chamada a funções exportadas nessa mesma instância. A instanciação do módulo engloba a inicialização do estado e "pilha" do programa, a inicialização das definições globais, memórias e tabelas, e a execução da função inicial do módulo. Esta última fase é executada dentro do ambiente onde o programa é incorporado. Na Figura 9, retirada de (Atapattu, 2020), encontra-se ilustrado este processo, onde está a ser comparado com as fases necessárias para executar código JS. Além disso, ainda ilustra o processo que costuma ser utilizado nas aplicações *web* que recorrem ao WASM, onde existe um servidor que prepara o código WASM a enviar para o lado do cliente.

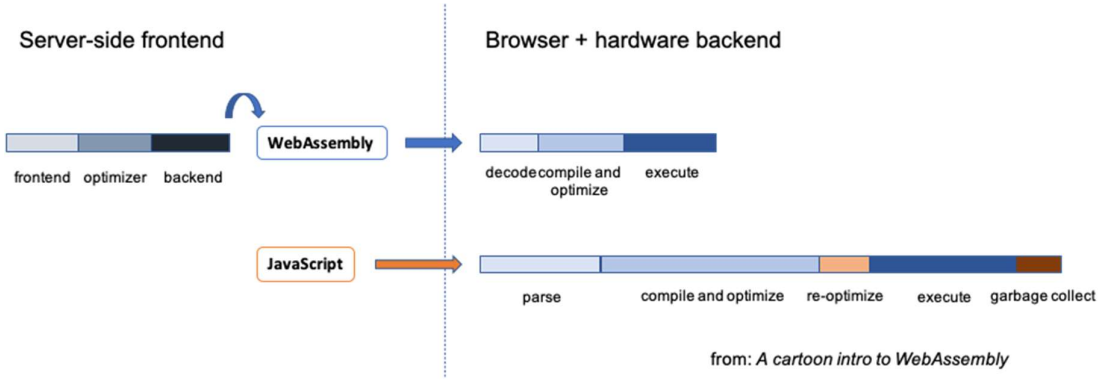


Figura 9 – Fases da semântica do WASM (Atapattu, 2020)

2.3. Estrutura

2.3.1. Módulo

Os programas WASM são organizados em módulos, que basicamente são a unidade de implementação, carregamento e compilação do código. Estes módulos declaram um conjunto de definições que poderão ser importadas ou exportadas para o ambiente hospedeiro.

As definições existentes englobam as funções e os respetivos tipos associados, tabela e respetivos elementos, memória linear e respetivos dados iniciais, função inicial, e instruções de importação e exportação de elementos. Cada classe de definição possui o seu próprio espaço de índices, que serve para os referenciar cada definição no respetivo módulo.

Os tipos no módulo definem a configuração de uma dada função, apresentando os tipos dos parâmetros e o respetivo resultado. Na Figura 10 foram definidos dois tipos de funções, o `$t0` que define uma função com dois parâmetros do tipo inteiro *32-bit* e retorna um valor do mesmo tipo, e o `$t1` que define uma função sem parâmetros e com o retorno do tipo inteiro *32-bit*.

```
(module
  (type $t0 (func (param i32) (result i32)))
  (type $t1 (func (result i32)))
)
```

Figura 10 – Código WAT com um exemplo da definição do elemento tipo

As funções estão sempre associadas a um respetivo tipo, e de acordo com o *scope* onde estão inseridas podem ou não possuir um "corpo" composto por instruções que manipulam a pilha. Desta forma, na Figura 11 foram definidas três funções distintas, a função `$transf` que é exportada com o nome "transform", a função `$letter` que é uma função interna do módulo, e a função `$caps` que é importada do módulo "env" no campo "caps". Uma vez que a função `$caps` é importada no módulo, a sua definição deve estar implementada no lado do hospedeiro.

```
(module
  (import "env" "caps" (func $caps (type $t1)))
  (type $t0 (func (param i32) (result i32)))
  (type $t1 (func (result i32)))
  (func $transf (type $t0) (param $i i32) (result i32)
    (call_indirect $T0 (type $t0)
      (local.get $i)
    )
  )
  (func $letter (type $t0) (param $i i32) (result i32)
    (block $B0
      (br_if $B0
        (i32.eqz
          (call $caps)
        )
      )
      (return
        (i32.load8_u offset=26
          (local.get $i)
        )
      )
    )
    (i32.load8_u
      (local.get $i)
    )
  )
  (export "transform" (func $transf))
)
```

Figura 11 – Código WAT com um exemplo da definição do elemento função

O código no corpo da função `$transf` invoca de forma indireta a função `$letter` com recurso à tabela de funções. A definição da tabela está ilustrada na Figura 12 onde também se encontra expressa a sua inicialização com a respetiva referência da função.

```
(module
  ...
  (table $T0 1 anyfunc)
  (elem $T0 (i32.const 0) $letter)
  ... ;; elemento export fica no fim.
)
```

Figura 12 – Código WAT com um exemplo da definição dos elementos tabela e segmentos de dados

A função `$letter` acede ao endereço de memória referente ao índice da letra passada nos argumentos. Caso a chamada à função `$caps` seja diferente de zero, significa que está a ser requerida a letra maiúscula, e por isso é feito o deslocamento para o endereço de memória onde se encontram definidas as letras maiúsculas. A definição e inicialização da memória encontra-se ilustrada na Figura 13.

```
(module
  ...
  (memory $M0 10 10)
  (data $M0 (i32.const 0) "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
")
  ...
)
```

Figura 13 – Código WAT com um exemplo da definição dos elementos memória e segmentos de dados

O exemplo completo está ilustrado na Figura 14 e consiste num programa WASM que faz a conversão de um número referente à posição de uma letra no alfabeto (A a Z – índice de 0 a 25). Opcionalmente pode ser devolvida uma letra minúscula ou maiúscula.

```
(module
  (import "env" "caps" (func $caps (type $t1)))
  (type $t0 (func (param i32) (result i32)))
  (type $t1 (func (result i32)))
  (func $transf (type $t0) (param $i i32) (result i32)
    (call_indirect $T0 (type $t0)
      (local.get $i)
    )
  )
  (func $letter (type $t0) (param $i i32) (result i32)
    (block $B0
      (br_if $B0
        (i32.eqz
          (call $caps)
        )
      )
      (return
        (i32.load8_u offset=26
          (local.get $i)
        )
      )
    )
    (i32.load8_u
      (local.get $i)
    )
  )
  (memory $M0 10 10)
  (data $M0 (i32.const 0) "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
")
  (table $T0 1 anyfunc)
  (elem $T0 (i32.const 0) $letter)
  (export "transform" (func $transf))
)
```

Figura 14 – Código WAT com um exemplo de um módulo WASM

Para complementar esta informação, encontram-se descritas na Tabela 1 as principais propriedades das definições presentes no módulo.

Tabela 1 – Propriedades dos componentes do módulo WASM

Nome	***Descrição***
Tipos	O componente *type* do módulo define um tipo de função. Este tipo pode ser declarado em mais que uma função.
Funções	O componente *func* do módulo define uma função. Esta função contém um determinado tipo definido através do componente *type*, e é composta por um conjunto de parâmetros, tipo de retorno, declaração de variáveis locais e o respetivo corpo da função. Para além disto, é possível declarar uma função inicial, sendo necessário associar a respetiva função à definição do componente *start*. Esta função é automaticamente invocada na instanciação do módulo, após a inicialização das tabelas e memórias.
	O corpo da função é composto por uma sequência de instruções que manipulam os valores da "pilha". Ao terminar a função, a "pilha" deve conter o valor que coincide com o tipo de resultado definido na função.
	As variáveis locais são referenciadas no corpo da função através do respetivo índice, sendo que para cada função, os parâmetros encontram-se ordenados nas primeiras posições do espaço.
	As funções são referenciadas através de índices presentes no respetivo espaço de funções, sendo que as funções importadas encontram-se ordenadas nas primeiras posições do espaço.
Tabelas e Segmentos de Elementos	O componente *table* é instanciado com um valor mínimo que especifica o tamanho inicial da tabela, e um valor máximo opcional que especifica o tamanho máximo que a tabela pode crescer. As tabelas correspondem a um vetor opaco de valores de um tipo específico de elemento, que podem ser acedidos através do respetivo índice.
	Os valores da tabela podem ser inicializados através de segmentos de elementos, definidos através do componente *elem*, ou externamente pelo ambiente hospedeiro. Este inicializa um subconjunto da tabela num dado deslocamento, através de um vetor estático de elementos. Quando a tabela não é inicializada encontra-se num estado vazio.
	As tabelas são referenciadas através de índices presentes no respetivo espaço de tabelas, onde tal como nas funções, começa-se sempre pelo menor índice. Atualmente o número máximo de tabelas por módulo é um.
Memórias e Segmentos de Dados	Uma memória é definida através do componente *memory*, e consiste num vetor contíguo de *bytes*. São instanciados com limites, sendo que o limite mínimo representa o tamanho inicial da memória e é obrigatório, já o tamanho máximo que esta pode alcançar é um valor opcional.
	A sua inicialização pode ser feita através de segmentos de dados, através do componente *data*, ou externamente pelo ambiente hospedeiro. Este inicializa um intervalo de memória num dado deslocamento, através de um vetor estático de *bytes*. Quando a memória não é inicializada, esta é automaticamente inicializada com o tamanho definido e os valores preenchidos a zero.

Nome	*Descrição*
	As memórias são referenciadas através de índices presentes no respetivo espaço de memórias, onde se começa sempre pelo menor índice. Atualmente o número máximo de memórias por módulo é um.
Variáveis Globais	O componente associado às variáveis globais é chamado *global*. Este armazena um determinado valor com um tipo predefinido, podendo este ser mutável ou imutável. A definição pode ser acompanhada pelo respetivo valor inicial. Caso este valor não seja definido, a variável assume o valor predefinido, que neste caso é sempre zero.
	As variáveis globais são referenciadas através de índices presentes no respetivo espaço de variáveis globais, onde se começa sempre pelo menor índice. Esta ordem não tem em conta as variáveis importadas.
Exportação	O componente *export* permite que um determinado elemento do módulo fique acessível no ambiente hospedeiro. Com isto, o acesso a estes elementos pode ser feito imediatamente após o módulo ter sido instanciado. Um elemento exportado deve ser rotulado através de um nome único que serve como chave para acesso externo.
	As definições dos elementos que podem ser exportados são funções, tabelas, memórias, e variáveis globais.
Importação	O componente *import* permite importar um determinado elemento para que fique acessível internamente ao módulo. Para isso as definições devem ser rotuladas através do nome do módulo onde estão incluídas, e do seu nome dentro desse módulo. Ao contrário das exportações, que necessitam obrigatoriamente de um identificador único, nas importações podem existir várias definições para o mesmo elemento desde que possuam índices distintos.
	As definições dos elementos que podem ser importados são funções, tabelas, memórias, e variáveis globais. A sua definição deve conter obrigatoriamente a referência para um destes elementos.
	As importações são referenciadas através de índices no respetivo espaço de *imports*, e vêm sempre ordenados antes do primeiro índice de qualquer outra definição existente no módulo.

As definições são referenciadas através de índices. Cada classe de definição possui o seu próprio espaço de índices, sendo que existe um espaço específico para cada uma das funções, onde não só existem as variáveis locais, como também são incluídos os argumentos passados à função. As instruções de controlo também possuem o seu próprio espaço, onde as *labels* são referenciadas pelo respetivo índice. Os espaços de índices estão organizados de forma sequencial e ordenada.

O exemplo da Figura 15, baseado no artigo (Sendilkumarn, 2020), apresenta o código de um programa WASM utilizado para gerar uma série Fibonacci. O código é apresentado no formato textual WAT, onde as instruções são representadas como uma *S-expression* (Nikolaev, 2021), isto é, uma expressão simbólica utilizada para representação de dados em dados em árvore e lista através do uso de parênteses - os exemplos de código WASM apresentados neste documento são apresentados sempre neste formato.

```
(module
  (type $t0 (func (param i32) (result i32)))
  (func $fib (type $t0) (param $p0 i32) (result i32)
    (local $l1 i32)
    (local.set $l1
      (i32.const 1))
    (block $B0
      (br_if $B0
        (i32.lt_s
          (local.get $p0)
          (i32.const 2)))
      (local.set $l1
        (i32.const 1))
      (loop $L1
        (local.set $l1
          (i32.add
            (call $fib
              (i32.add
                (local.get $p0)
                (i32.const -1)))
            (local.get $l1)))
        (br_if $L1
          (i32.gt_s
            (local.tee $p0
              (i32.add
                (local.get $p0)
                (i32.const -2)))
            (i32.const 1)))))
    (local.get $l1))
  (export "fib" (func $fib)))
```

Figura 15 - Módulo WASM para gerar série Fibonacci artigo (Sendilkumarn, 2020)

Tal como é feito para qualquer programa WASM, este exemplo é composto por um módulo que contém todas as definições do programa. Neste módulo está definida uma função que calcula a série Fibonacci. Esta função possui o índice `$fib` e implementa o tipo `$t0`, ou seja, recebe um parâmetro do tipo inteiro *32-bit* (neste caso, acedido através do índice `$p0`) e devolve um valor também do tipo inteiro *32-bit*. A função é exportada com o nome `fib`.

Na função `$fib` existem diversos tipos de instruções tais como, instruções numéricas (por exemplo, "`(i32.add ... ...)`"), acessos a variáveis locais (por exemplo, "`(local.get ...)`"), e instruções de controlo (por exemplo, "`(loop ...)`") (ver Secção 2.3.2). Esta função é recursiva como pode ser verificado através da instrução "`(call $fib ...)`" presente no código.

2.3.2. Instruções

Como foi mencionado na Secção 2.3.1, o código WASM é formado por instruções que manipulam valores na "pilha" e podem ser divididas em instruções simples e instruções de controlo.

As instruções simples porventura podem ainda ser subdivididas em instruções numéricas, paramétricas, variáveis globais ou locais, de memória e em expressões.

As instruções numéricas (Figura 16) são formadas por declarações de constantes, onde é definida uma constante estática, operações unitárias e binárias que consomem um ou dois operandos e produzem um resultado, operações de teste, isto é, consomem um operando de um determinado tipo e produzem um resultado booleano, comparações entre dois operandos do mesmo tipo, e conversões de tipo.

```
(i32.const ...) ;; criação de constante estática () : (i32)
(f32.ceil (...)) ;; operação unitária de arredondamento (f32) : (f32)
(i32.add (...) (...)) ;; operação binária de adição (i32, i32) : (i32)
```

Figura 16 - Código WAT com exemplos de instruções numéricas

Depois existem as instruções paramétricas (Figura 17), que são instruções que aceitam qualquer tipo de dados. Apesar de consumirem qualquer tipo de operando, estas possuem restrições no número de operandos consumidos, e no tipo de resultado produzido.

```
(select (...) (...) (...)) ;; instrução paramétrica select (?, ?, i32) : (?)
(drop (...)) ;; instrução paramétrica drop (?) : ()
```

Figura 17 - Código WAT com exemplos de instruções paramétricas

Relativamente às instruções para acesso a variáveis (Figura 18), estas são responsáveis por obter ou alterar o valor de uma determinada variável local ou global. Para isso, é necessário recorrer ao índice da variável, que para variáveis locais encontra-se incluído no espaço de índices da função, e para uma variáveis globais num espaço próprio de índices que é partilhado pelas várias funções do módulo. O tipo de operandos e resultados não só varia com a instrução como também com o tipo da variável em questão.

```
(local.get ...) ;; obter variável local () : (?)
(local.set ... (...)) ;; alterar variável local (?) : ()
(local.tee ... (...)) ;; alterar e obter variável local (?) : (?)
(global.get...) ;; obter variável global () : (?)
(global.set ... (...)) ;; alterar variável global (?) : ()
```

Figura 18 - Código WAT com exemplos de instruções para acesso a variáveis

As instruções de memória são organizadas em dois tipos distintos: instruções para carregar e armazenar valores (Figura 19), e instruções de gestão de memória (Gohman, Lepesme, Qwerty2501, Spencer, & Um, 2021) (Figura 20). As instruções para carregar e armazenar valores na memória requerem sempre um argumento que contém o deslocamento do endereço e outro com o alinhamento esperado (expresso como o expoente de uma potência de 2). Estes argumentos podem ser omitidos, uma vez que por predefinição são preenchidos com o valor zero. Estas instruções podem especificar um tamanho de armazenamento menor do que o tamanho do respetivo tipo de valor passado como operando (por exemplo, `i32.load8_s`). Em relação às instruções de gestão de memória, existe uma instrução que

devolve o tamanho atual da memória no módulo, e outra instrução que permite aumentar o tamanho dessa memória.

```
(i32.load offset=... (...)) ;; obter valor na memória (i32) : (i32)
(i32.load8_s (...)) ;; obter valor de 8 bits na memória (i32) : (i32)
(i32.store (...) (...)) ;; armazenar valor na memória (i32) : (i32)
```

Figura 19 - Código WAT com exemplos de instruções de acesso à memória

```
(memory.grow ... (...)) ;; aumenta a memória (i32) : (i32)
(memory.size ...) ;; devolve o tamanho da memória () : (i32)
```

Figura 20 - Código WAT com exemplos de instruções de gestãoo de memória

Por último, dentro do conjunto das instruções simples, existem as expressões (Figura 21), que corresponde a um conjunto de instruções utilizadas no corpo das funções, na inicialização de valores para variáveis globais, e na declaração de deslocamentos dos segmentos de dados e elementos das tabelas.

```
(local.set ... (i32.const ...)) ;; expressão no corpo da função
(global ... (...) (...) (i32.const ...)) ;; expressão na definição de uma
variável global
(data ... (f32.const ...) ...) ;; expressão no segmento de dados
(elem ... (i32.const ...) ...) ;; expressão nos elementos da tabela
```

Figura 21 - Código WAT com exemplos de expressões

As instruções de controlo (Figura 22) têm como função realizar alterações no fluxo de execução, permitindo assim a criação de blocos de código, ciclos, chamadas a funções, retorno de valores, execução condicional, ramificação da execução, etc. É possível identificar dois grupos dentro deste tipo de instruções: instruções estruturadas e de ramificação. Uma instrução estruturada pode ser um bloco de código, ciclo ou condição, e não só termina sempre com uma pseudo-instrução (por exemplo, `end`), como também introduz uma *label* (implícita ou explícita) que os transforma em alvos para as instruções de ramificação. As instruções de ramificação podem ser condicionais ou não, e permitem que a execução seja transferida para uma dada instrução de controlo rotulada por uma *label*. As chamadas invocam uma função, consumindo os argumentos necessários da "pilha" e produzindo os valores do resultado.

```
(block $B0
  (br_if $B0 (...))
  (loop $L1
    (call ...)
    (br_if $L1 (...)))) ;; instruções de controlo
```

Figura 22 - Código WAT com exemplos de instruções de controlo

2.4. Máquina Virtual do WebAssembly

A máquina virtual do WASM é suportada nos vários navegadores modernos, uma vez que foi integrada na máquina virtual do JS, oferecendo à maioria das aplicações *web* a possibilidade de obterem melhores desempenhos através da utilização do WASM (Gear Technologies, 2021).

No entanto, a integração de um novo módulo na máquina virtual do navegador, implementado quase de raiz uma vez que requer um novo formato binário, pode levantar diversas questões tais como, o porquê de não terem sido utilizadas linguagens de programação que já conseguem ser executadas em diversos ambientes, como por exemplo Java através da Java Virtual Machine (JVM) (Team O. , JVM, 2021), e C# (Team M. , C# documentation, 2021) que através da Common Language Runtime (CLR) (Team M. , CLR overview, 2021) suporta o padrão Common Language Infrastructure (CLI), ou o porquê de ter sido implementado um novo formato binário, com um conjunto de instruções específico para a sua Instruction Set Architecture (ISA), invés de ter sido utilizado o código binário produzido por compiladores como o LLVM (Team L. A., 2021).

A principal razão pela qual não foi aproveitado nenhum outro formato de instruções já existente foi o facto de nenhum destes possuir a capacidade de atingir os desempenhos pretendidos. Isto porque, por exemplo, enquanto que o WASM foi projetado para facilitar o processo de validação, possuindo um fluxo de controlo estruturado e um processo de compilação e validação de passagem única, os restantes formatos de instruções requerem mais tempo e processamento para realizarem essas tarefas (Atapattu, 2020).

Para além disso, não existia qualquer formato que satisfizesse os requisitos e objetivos estabelecidos. O desejado é que o conjunto de instruções seja o mesmo independentemente da arquitetura de máquina, isto é, que seja estável e não varie ao longo do tempo. A sua representação também deve ser a mais pequena possível, de forma a otimizar a sua transmissão pela Internet, e ainda ser suficientemente eficiente na descompressão, decodificação e compilação dos programas para uma rápida inicialização. Por fim, o comportamento dos programas deve ser o mais previsível e determinista possível, independentemente da arquitetura onde executa. Nenhum dos formatos existentes cumpria com estes requisitos (Community W. , 2020).

Contudo, de acordo com a equipa de desenvolvimento do WASM, é possível adaptar um dos compiladores existentes (por exemplo, o LLVM) para executar na *web*, uma vez que nenhum dos problemas existentes é intransponível. Em (Community W. , 2020), esta equipa indica que tal já foi feito para outras linguagens, "*por exemplo, o PNaCl define um pequeno subconjunto portátil da representação intermédia (IR) com comportamento indefinido reduzido e uma versão estável da codificação de código de bits, empregando também várias técnicas para melhorar o desempenho da inicialização*", no entanto, para eles "*cada personalização, solução alternativa e solução especial significa menos benefícios da infraestrutura comum utilizada*". Desta forma, defendem que é melhor projetar um formato

binário melhorado para os objetivos e requisitos identificados acima, ao invés de adaptar um sistema projetado para outros fins.

Com base em (Fredriksson, 2020), é possível afirmar que o WASM é a melhor tecnologia em termos de desempenho e segurança que existe atualmente para ser executada na *web*. Nesta afirmação exclui-se o Portable Native Client (PNaCl), que apesar de ser ligeiramente mais rápido que o WASM, segundo (Krill, 2017), este foi descontinuado por falta de adesão por parte dos criadores de navegadores. Ainda em (Fredriksson, 2020), conclui-se que o WASM não só é o mais rápido, como também o mais seguro, uma vez que o ActiveX (Chernysh, 2015) e os Java applets (Team O. , Java Applets, 2021) exibem vulnerabilidades que permitem a uma aplicação ter a capacidade de obter acesso completo à máquina do utilizador, tornando-os assim não recomendados para a *web*.

2.5. Interação com o JS

O WASM foi projetado para ser um complemento e não uma substituição do JS, sendo criado para melhorar o desempenho das aplicações *web*, uma vez que é baseado num formato binário compacto e altamente eficiente. Desta forma, WASM e JS são interoperáveis, partilhando entre si funções, memórias, variáveis e objetos. Esta comunicação é bidirecional uma vez que não só podem ser importados cada um destes elementos para o módulo, como podem ser exportados para o JS (Community W. , 2020) (Hernandez, 2021).

Para que fosse possível a integração do WASM no JS foi disponibilizado um API que permite não só instanciar módulos WASM no código JS, como também partilhar elementos e alterar as propriedades de alguns desses elementos (por exemplo, alterar o tamanho da memória). O objeto `WebAssembly` no JS atua como um *namespace* para todas as funcionalidades relacionadas ao WASM. De acordo com o (MDN Contributors, WebAssembly, 2021), os principais casos de uso deste objeto são:

- Carregamendo do código WASM através das funções `WebAssembly.instantiate` e `WebAssembly.instantiateStreaming`.
- Criação de novas instâncias de memória e tabela através dos construtores `WebAssembly.Memory` e `WebAssembly.Table`, respetivamente.
- Fornecer recursos para lidar com erros que ocorram no WASM através dos construtores `WebAssembly.CompileError`, `WebAssembly.LinkError` e `WebAssembly.RuntimeError`.

A seguir vão ser demonstrados os passos necessários para realizar a integração de um módulo WASM com uma aplicação JS. Para isso, vão ser apresentados alguns exemplos de código retirados e adaptados de (ICHI.PRO, 2021).

O primeiro passo na integração do módulo com uma aplicação JS, ilustrado na Figura 23, passa por carregar o ficheiro WASM como um recurso e instanciar o módulo com os dados carregados. Assim, começa-se por carregar o ficheiro através da chamada à função `fetch` (MDN Contributors, Using Fetch, 2021), sendo que a resposta deve ser convertida num *buffer* de dados binários (MDN Contributors, ArrayBuffer, 2021). Por fim, depois de ter o

módulo carregado num *buffer* binário, deve ser feita a sua instanciação através do método `instantiate` (MDN Contributors, WebAssembly.instantiate(), 2021) presente no objeto `WebAssembly`. Nota que esta não é a forma mais eficiente de ser feito a instanciação do módulo, e por isso, na Figura 24 encontra-se ilustrado o código que, segundo a documentação do Mozilla (MDN Contributors, WebAssembly, 2021), é mais eficiente uma vez que compila e instancia um módulo WASM diretamente de um *streaming* de dados (MDN Contributors, WebAssembly.instantiateStreaming(), 2021).

```
async function loadWasm(file, imports) {
    let bytes = await fetch(file);
    let bytesBuffer = await bytes.arrayBuffer();
    let wasmModule = await WebAssembly.instantiate(bytesBuffer, imports);
    return wasmModule;
}
let module;
(async function() {
    module = await loadWasm('main.wasm'); // main.wasm é o ficheiro WASM com o
módulo a integrar.
})()
```

Figura 23 - Código JS com a inicialização do módulo WASM (ICHI.PRO, 2021)

```
async function loadWasm(file, imports) {
    let wasmModule = await WebAssembly.instantiateStreaming(fetch(file), import
s);
    return wasmModule;
}
```

Figura 24 - Código JS com o carregamento do módulo WASM através do método *instantiateStreaming*

Para conseguir importar uma função JS no módulo é necessário incluir a definição da função no objeto passado durante a instanciação do módulo. Na Figura 25 encontra-se ilustrado o código para o objeto importado no módulo, onde é incluída a função `log` que por sua vez invoca a função JS `console.log` (MDN Contributors, console.log(), 2021) que imprime na consola o valor passado como argumento. Esta função encontra-se incluída no módulo de importação chamado `js`.

```
// ...
let module;
(async function() {
    let imports = {
        js: {
            log: (number) => console.log(number)
        }
    };
    module = await loadWasm('main.wasm', imports); // main.wasm é o ficheiro WA
SM com o módulo a integrar.
})()
```

Figura 25 - Código JS com importação do objeto no módulo WASM (ICHI.PRO, 2021)

No módulo WASM deve existir a respetiva declaração correspondente à função `log`. Na Figura 26 encontra-se expressa essa definição, onde é declarada com o índice `$log` (ou o índice zero no seu espaço de índices). Para além disso, está ilustrada uma função (`$printNumber`) que é responsável por chamar a função importada no módulo. Esta recebe um parâmetro do tipo inteiro *32-bit* e invoca a função `log` com o valor desse parâmetro.

```
(module
    (import "js" "log" (func $log (param i32)))
    (func $printNumber (param $number i32)
        (call $log
            (local.get $number)
        )
    )
)
```

Figura 26 - Código WASM com a definição da função importada *log* (ICHI.PRO, 2021)

Como foi mencionado acima, a comunicação entre WASM e JS pode ser bidirecional. Desta forma, para utilizar a função `$printNumber` no lado do JS é necessário incluir a definição de exportação no módulo. Depois disso, a função está disponível para ser invocada no JS, através do nome indicado na declaração. Na Figura 27 está ilustrado a definição incluída no WASM, e na Figura 28 a chamada à função de acordo com a definição no módulo. Os elementos exportados encontram-se sempre inseridos no objeto `exports` da instância do módulo no JS, e neste caso, a função como foi exportada com o nome `printNumber`, deve ser invocada através da chamada `exports.printNumber`.

```
(module
    ...
    (export "printNumber" (func $printNumber))
)
```

Figura 27 - Código WASM com a definição da função *printNumber* exportada (ICHI.PRO, 2021)

```
// ...
function printNumber(number) {
    module.instance.exports.printNumber(number);
}
```

Figura 28 - Código JS com a chamada à função exportada *printNumber* do módulo (ICHI.PRO, 2021)

De acordo com (Clark, 2018), esta comunicação nem sempre foi tão eficiente. Contudo, após a otimização descrita nesse artigo, as chamadas entre WASM e JS tornaram-se extremamente eficientes, ficando ainda mais rápidas do que chamadas efetuadas a funções *non-inlined* (Rakshit, 2019) dentro do JS.

Atualmente, a maioria dos navegadores mais conhecidos suportam WASM. Com base na página *web* (Can I Use, 2021), no dia 17 de dezembro de 2021, estes navegadores são:

- Edge – Versão 16 ou mais recente
- Firefox – Versão 52 ou mais recente
- Chrome - Versão 57 ou mais recente
- Safari - Versão 11 ou mais recente
- Opera – Versão 44 ou mais recente

Apesar disto, a compatibilidade apresentada na documentação do Mozilla (MDN Contributors, WebAssembly, 2021) apresenta algumas funções do API disponibilizado que poderão não ser suportadas pelos mesmos navegadores.

CAPÍTULO 3.
TRABALHO RELACIONADO

WASM é uma tecnologia relativamente recente e por isso possui um ecossistema em constante expansão, onde o desenvolvimento de novas ferramentas é acelerado, existindo novidades quase todos os meses. No entanto, atualmente existem poucas ferramentas para manipulação de código, sendo que a maioria está relacionada apenas com a análise do mesmo.

Neste capítulo serão abordadas ferramentas já existentes no ecossistema do WASM que possuem um objetivo muito semelhante ao da ferramenta tratada no presente trabalho (Secção 3.1). Com isto, pretende-se não só compreender até que ponto essas ferramentas são capazes de solucionar o problema exposto, como também fazer uma comparação dessas ferramentas com o WasmManipulator. Para além disso, haverá um secção (Secção 3.2) com a análise de algumas das linguagens de instrumentação existentes no mercado, que serviram como base na implementação do mecanismo de transformação de código na ferramenta. Por fim, existirá uma secção (Secção 3.3) com as linguagens de pesquisa e substituição de código que foram consideradas para implementar especificamente a funcionalidade de pesquisa por padrão(funcionalidade *templates* descrita na Secção 3.3 do Anexo B).

3.1. Tecnologias Existentes

3.1.1. Wasabi

Segundo (Lehmann & Pradel, 2018) o Wasabi consiste na primeira ferramenta de uso geral para análise dinâmica de código WASM. Esta permite facilmente implementar um conjunto vasto de análises, que por sua vez permitem a monitorização do comportamento de baixo nível de um programa. A transformação é feita ao nível do binário, onde é inserido todo o código WASM necessário para que durante a execução sejam chamadas as respetivas funções de análise. Para realizar esta análise, a ferramenta implementa estas funções de análise no lado do JS, precisando assim de gerar um código JS que servirá de auxílio ao módulo.

Para aplicar as transformações, a ferramenta disponibiliza um conjunto de *hooks*, que representam as funções de análise (implementadas no JS) chamadas sempre que uma dada instrução é executada. Esses *hooks* carregam informação sobre a respetiva instrução executada, estando esta disponível para acesso no código JS do utilizador.

Esta ferramenta é unicamente focada na instrumentação e análise dinâmica de execução, e por isso, quando se trata de transformações de código, é demasiado limitada comparado com o que é pretendido. Para além disso, a ferramenta para realizar a análise baseia-se na injeção de código JS (relativamente extenso) invocado em cada uma das diversas instruções existentes no código, e como consequência afeta negativamente o desempenho do programa.

Sabendo isto, percebe-se que as funcionalidades presentes na ferramenta não só poderão ser completamente substituída pela ferramenta WasmManipulator, uma vez que esta também deverá permitir a instrumentação e análise dinâmica da execução, até mesmo com maior

eficiência uma vez que muito do código JS pode ser removido, como também oferecerá maior flexibilidade ao permitir muitas outras formas de manipulação do código no módulo.

3.1.2. WasmProf

Conforme descrito em (Gifford, 2019), WasmProf tem como principal objetivo monitorar de forma precisa a execução de funções no módulo WASM. Este objetivo é muito semelhante ao objetivo do Wasabi (descrito acima), no entanto, não só possui um âmbito muito mais reduzido, limitando-se à monitorização de funções e chamadas a funções, como também pretende remover parte da sobrecarga existente com a injeção do JS, uma vez que o código JS existente para este *profiler* foi reescrito de uma forma mais eficiente e amigável para o compilador JS, e por isso, o impacto que esta tem no desempenho do programa WASM é muito menor.

Basicamente a ferramenta aplica as alterações nos arcos de chamadas, isto é, regista não só a instrução de chamada como a função invocada, mantendo um contador com o número de vezes que determinado arco foi percorrido e quanto tempo foi gasto nesse arco. Para registar o tempo, será necessário a injeção de algum código JS, sendo que uma das funções implementada neste código retorna a data atual num formato numérico, e as restantes alteram os registos dos arcos.

Tal como o Wasabi, esta poderá ser completamente substituída pela ferramenta WasmManipulator. Assim sendo, o único aspeto que pode ser vantajoso no WasmProf é a sua eficiência, uma vez que, ao contrário do WasmManipulator, este foi desenhado para esse efeito. Mesmo assim, para afirmar tal vantagem, teria de ser feita a devida comparação, uma vez que o mecanismo de transformação de ambas as ferramentas poderá ser implementado de forma semelhante.

3.1.3. WABT

Segundo a documentação da ferramenta (Smith & Community, 2021), WABT (WebAssembly Binary Toolkit) é um agregador de ferramentas utilizadas como auxílio na manipulação de código WASM. As ferramentas disponibilizadas englobam a conversão de código WASM para WAT e vice-versa, a remoção de secções no código WASM, entre outras.

Como o nível de transformações oferecido pelas ferramentas é muito superficial, estas ferramentas são maioritariamente destinadas à integração noutros sistemas que pretendam realizar manipulações ao código WASM de maior complexidade. Desta forma, não só as ferramentas foram projetadas para serem simples e eficientes, como também para serem de fácil integração. A execução destas ferramentas procura seguir com rigor as especificações da linguagem WASM, e por isso, não oferecem qualquer otimização ou compilação de alto-nível.

Apesar do WABT ser uma ferramenta que serve exclusivamente para manipular código WASM, o seu objetivo está muito aquém dos objetivos definidos para este trabalho.

Contudo, esta possui ferramentas que poderão contribuir diretamente para o desenvolvimento do WasmManipulator.

3.1.4. WAIL

A ferramenta WebAssembly Instrumentation Library (WAIL), desenvolvida por Jack Baker, consiste numa biblioteca JS utilizada para realizar modificações no código WASM. Com base em (Baker, 2019), a ferramenta foi desenvolvida principalmente para analisar e modificar código WASM para videojogos. Para isso, foram definidos os seguintes requisitos para a biblioteca: as transformações devem ser realizadas dentro do navegador, isto é, a manipulação deve ser feita diretamente no JS; e o processo de manipulação deve ser extremamente eficiente, uma vez que o código para este tipo de programas tende sempre a ser muito extenso, e como consequência pode afetar drasticamente o seu desempenho.

Para permitir que as transformações fossem realizadas o mais rápido e eficiente possível, WAIL não só faz o *parse* apenas das secções e elementos que são necessários para a modificação, como também recorre a *streams* para tratar e modificar cada elemento assim que é lido, diminuindo consideravelmente a quantidade de memória necessária para executar, uma vez que não necessita de armazenar as informações dos elementos transformados. Para além disso, também precisou de implementar um API relativamente rígido, disponibilizando um leque bastante limitado de métodos bem definidos, que apenas permitem adicionar e editar elementos individualmente, adicionar e remover secções, e aceder a variáveis globais e funções através do respetivo índice interno.

Esta abordagem exigiu que determinadas modificações precisassem de mais do que uma chamada ao API. Estas chamadas devem ser feitas de uma forma sequencial, passando conhecimento umas às outras. Por exemplo, para adicionar uma função é necessário que primeiro seja inserido o elemento *type* (tipo da função), que por sua vez devolve uma referência para esse elemento. Essa referência deve então ser passada na próxima chamada, que consiste na inserção do elemento *func* (definição da função), que tal como o tipo da função, devolve uma referência para esse elemento. Com essa referência é possível inserir o código para a função através da chamada ao respetivo método.

Desta forma, é possível concluir que apesar da ferramenta ser ideal quando as transformações têm de ser feitas diretamente no JS e/ou o conteúdo transformado é extremamente extenso, esta é relativamente limitada no que toca à variedade e ao modo como as transformações são aplicadas. Assim, não só existe pouca flexibilidade, como também a complexidade aumenta consideravelmente quando as transformações são também algo complexas. Estas propriedades são o oposto das existentes na ferramenta WasmManipulator, uma vez que este, apesar de ser relativamente eficiente, quando o código é demasiado extenso poderá sofrer quebras não só de desempenho como de memória. Esta também não permite que a transformação seja realizada diretamente no JS, precisando que as devidas transformações sejam executadas localmente, para que depois o módulo resultante seja integrado no JS. Contudo, quando a complexidade das transformações é

elevada e o nível de flexibilidade e abstração é o aspeto fulcral requerido pelo utilizador, o WasmManipulator é a opção indicada.

3.1.5. Análise Comparativa

Na Tabela 2 encontra-se um resumo da análise realizada para as diferentes ferramentas abordadas nesta secção.

Tabela 2 – Análise comparativa das tecnologias existentes

	Wasabi	***WasmProf***	***WABT***	***WAIL***
Função	Instrumentação	Instrumentação	Transformação	Transformação
Objetivo	Análise dinâmica de código WASM	Monitorar a execução de funções no código WASM	Agregador de ferramentas para auxílio na manipulação de código WASM	Realizar modificações de forma eficiente no código WASM
Desempenho	Lento	Médio	Médio	Rápido
Complexidade	Fácil	Muito Fácil	Fácil	Difícil
Nível das Transformações	Muito Limitado	Muito Limitado	Limitado	Semi-Limitado
Ambiente de Execução	JS	Terminal/JS	Terminal	JS

O Wasabi é uma linguagem de instrumentação que tem como objetivo a análise dinâmica de código WASM. Ao implementar uma grande quantidade desta análise em código JS, faz com que este afete consideravelmente o desempenho do programa WASM. Contudo, esta forma de análise poderá simplificar a sua utilização, que é efetuada através de *callbacks* no lado do JS para as várias instruções do código. Sendo uma ferramenta de análise, significa que o nível de transformações disponibilizadas é muito limitado.

Tal como o Wasabi, WasmProf é uma linguagem de instrumentação, no entanto, permite apenas monitorar a execução de funções e respetivas chamadas no código WASM. Devido à sua especificidade, este reduz muito o código JS necessário para a análise, e por isso é mais rápido que o Wasabi. A sua complexidade é também menor, uma vez que a análise é feita de forma automática. O WasmProf é executado no terminal para realizar as transformações ao módulo, contudo, os resultados só são apresentados durante a execução do programa transformado (ambiente JS).

A ferramenta WABT é um agregador de múltiplas ferramentas que permitem auxiliar sistemas de manipulação de código WASM. Na generalidade o desempenho da ferramenta é bom, no entanto, um pouco limitado quando a dimensão do código é extremamente elevada. Cada uma destas ferramentas possuem objetivos bem definidos, e por isso, para realizar transformações complexas devem ser utilizadas juntamente com um sistema de manipulação. Estas são executadas através do terminal.

Relativamente ao WAIL, esta é uma ferramenta projetada especificamente para a transformação eficiente de código WASM realizada diretamente no ambiente JS. Para obter esta eficiência, a ferramenta não só limita a quantidade de transformações que disponibiliza, mas também obriga que o utilizador tenha conhecimento dos detalhes internos do programa WASM e da linguagem em si.

Como é possível verificar, de momento não existe nenhuma ferramenta que permita a realização de transformações ao código WASM de forma simples e flexível.

3.2. Linguagens de Transformação

Nesta secção são descritas tecnologias análogas as WasmManipulator, mas destinadas a outros alvos tecnológicos.

3.2.1. ASM

De acordo com o guia de utilizador (Bruneton, 2011), ASM é uma das muitas ferramentas destinada à geração e transformação de classes Java (Team O. , Java, 2021). Esta permite o funcionamento em tempo de execução, isto é, alteração de classes Java já compiladas para *bytecode*, mas também suporta o modo de funcionamento em tempo de compilação. Esta ferramenta foi também projetada para ser o mais rápida e pequena possível, uma vez que por funcionar em tempo de execução não deve afetar o desempenho das aplicações que o usam.

O ASM fornece um conjunto pré-definido de transformações e algoritmos de análise comuns, a partir dos quais podem ser utilizados em transformações ou ferramentas de análise de código mais complexas. Para realizar estas tarefas são disponibilizados dois APIs, onde o principal possui uma representação de classes baseada em eventos, baseado-se muito no padrão *Visitor*, enquanto que o outro fornece uma representação baseada em objetos.

O facto do ASM oferecer controlo total, onde as transformações são consideradas de baixo nível, leva a que a sua complexidade seja bastante elevada. Para além disso, é considerado o padrão da indústria para manipulação de *bytecode*, dificultando a transversalidade de conceitos.

3.2.2. Javassist

Javassist é uma ferramenta de manipulação de Java *bytecode* que disponibiliza um API de alto e baixo nível. Segundo a documentação (Long, 2021), esta permite a inspeção de programas Java, a criação de novas classes em tempo de execução, e a modificação de qualquer classe após o seu carregamento.

A inspeção é realizada de forma equivalente ao que é feito diretamente em Java por meio da API do *Reflection*. Contudo, a modificação das classes engloba um conjunto de classes próprias do Javassist. Uma dessas classes é responsável por rastrear e controlar as classes carregadas, utilizando para isso uma *pool* de classes. O carregamento das classes para a *pool* pode seguir o mesmo comportamento que o *class loader* do JVM, ou pode ser realizado através de um caminho definido pelo utilizador. Depois existem outras classes que representam campos, métodos e construtores. Estas classes disponibilizam um API

destinado à sua modificação, que inclui a inserção de código antes e depois do elemento, a inserção de um bloco de código que executa quando ocorre uma exceção (`catch`), entre outros. Para além disto, todas estas classes fornecem dados sobre os elementos que representam.

Apesar de ser possível replicar a abordagem seguida pelo Javassist, esta possui algumas desvantagens. Uma delas é a falta de flexibilidade, uma vez que as transformações não podem ser reutilizadas em mais que um ponto, ou seja, cada transformação é apenas aplicada num único elemento. Depois, o nível de transformações disponibilizadas é muito limitado, uma vez que só permite transformações antes, depois ou em torno de um método.

3.2.3. AspectJ

Segundo o livro de (Laddad, 2010), AspectJ (Foundation, 2021) consiste numa extensão orientada a aspetos para a linguagem de programação Java. O paradigma orientado a aspetos (Aspect-Oriented Programming - AOP) (Wang, Knutson, Donor, Botting, & Harris, 2014) pretende complementar outros paradigmas, como por exemplo a programação orientada a objetos (Object-Oriented Programming – OOP) (Doherty, 2020), permitindo encapsular "preocupações" transversais em componentes isolados e com um propósito específico (Separation of Concerns – SoC). Como consequência permite uma melhor modularização e abstração no código, facilitando a manutenção das aplicações (Dambekalns, Müller, Lemke, & Waidelich, 2021).

Desta forma, o AspectJ pode ser utilizado na implementação de diversas questões transversais, tais como verificação e tratamento de erros, sincronização, comportamento sensível ao contexto, otimizações de desempenho, monitorização, registo de *logs*, e suporte a *debug*. Isto tudo de forma modular e seguindo o princípio DRY (Don't Repeat Yourself) (Cneude, 2018), uma vez que para além do código ser declarado separadamente do código que será modificado, este encontra-se escrito num componente isolado.

Apesar disto, esta linguagem nem sempre possuiu esta abordagem, começando por ser apenas uma linguagem especial que permitia estender Java com novas *keywords*. Para isso possuía um compilador próprio que era capaz de interpretar e compilar essas extensões. Contudo, ao longo do tempo foi evoluindo, e atualmente passou a adotar uma linguagem semelhante ao Java. Isto não só facilita a transversalidade das alterações, como também já não é preciso a utilização de um compilador especial, uma vez que utiliza o compilador Java. Além disso, passou de uma linguagem que permitia pequenas extensões, para uma linguagem que permite realizar quase todo o tipo de transformações, incluindo a modificação de métodos estáticos, inserção de novos campos, inserção de *interfaces*, etc. (Laddad, 2010).

A aplicação das modificações tanto pode ser feita em tempo de execução, onde as alterações são introduzidas em *bytecode*, como em tempo de compilação, onde as alterações são introduzidas sobre o código-fonte.

O AspectJ permite a recolha de um conjunto de *join-points* no código do programa através de uma dada expressão (*pointcut*), e aplica a cada um desses pontos o código (*advice*)

definido pelo utilizador. Com isto, é possível definir os seguintes conceitos presentes nas linguagens orientadas aspetos, incluindo esta linguagem:

- *Join-Point* – define um ponto no programa onde o código do aspeto será integrado com o código da aplicação.
- *Pointcut* – representa uma expressão responsável pela captura desses *join-points*.
- *Advice* – representa todo o contexto envolvido na alteração do código associado ao *join-point*, isto é, representa não só o código a ser integrado na aplicação, mas também todos os dados relacionados com o *join-point* que se encontra a ser executado.

Resumidamente, o *advice* é o código que é executado a cada *join-point* obtido por um *pointcut* (Costa & Berkenbrock, 2003).

O exemplo da Figura 29, retirado do guia de desenvolvimento da linguagem AspectJ (AspectJ, 2001), ilustra a forma como estes conceitos se relacionam para que seja obtido o resultado final esperado. Neste exemplo, pretende-se fazer o rastreamento de três classes distintas, `TwoDShape`, `Circle` e `Square`, ao imprimir o antes e depois de quando estas são instanciadas, e de quando um dos seus métodos é invocado. Com isto, foram criados três *pointcuts*, o `myClass` que é responsável por obter o código executado dentro destas classes através da definição `within`, o `myConstructor` que, com recurso ao *pointcut* anterior, obtém o código do construtor da classe através da definição `execution(new(..))`, e por fim, o `myMethod` que tal como o anterior utiliza o *pointcut* `myClass`, mas ao adicionar a definição `execution(* *(..))`, obtém o código para todos os métodos existentes nestas classes. Depois, para cada um destes dois últimos *pointcuts* foi adicionado um ponto de entrada antes e depois do código. Por fim, dentro desses pontos foi inserido código para realizar o acesso à assinatura do método através dos dados contidos no objeto `thisJoinPointStaticPart`. O resultado, também retirado do mesmo guia, encontra-se ilustrado na Figura 30.

```
aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

Figura 29 - Código AspectJ com AOP aplicado a uma aplicação Java (AspectJ, 2001)

```
 --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  (…)
  --> double tracing.Circle.perimeter()
  <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
  (…)
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

Figura 30 - Resultado da execução das transformações realizada a uma aplicação Java com o AspectJ (AspectJ, 2001)

Com isto, conclui-se que esta ferramenta é bastante completa, seguindo com rigor os conceitos ditados pelo paradigma orientado a aspetos. Para além disto, percebe-se que a maioria dos conceitos aqui presentes são transversais a qualquer linguagem, e como exemplo, temos o AspectC++ (Spinczyk & GmbH, 2021) que consiste numa adaptação para a linguagem C++. Desta forma, decidiu-se que a ferramenta WasmManipulator deveria ser desenvolvida com base em princípios similares aos usados no AspectJ, usufruindo dos vários benefícios oferecidos pela abordagem tomada nessa linguagem, tais como, modularização e simplicidade.

3.2.4. Análise Comparativa

Na Tabela 3 encontra-se um resumo da análise realizada para as diferentes ferramentas abordadas nesta secção.

Tabela 3 - Análise comparativa das ferramentas de transformação para tecnologias análogas as WasmManipulator

	ASM	***Javassist***	***AspectJ***
Controlo	Total	Pouco	Elevado
Completude	Completo	Incompleto	Completo
Complexidade	Difícil	Médio	Fácil
Rapidez	Rápida	Média	Média
Transversalidade de Conceitos	Não (*bytecode*)	Sim	Sim
Tamanho	Grande	Pequeno	Médio

O ASM é uma ferramenta de baixo-nível para transformação de *bytecode*, projetada para funcionar tanto em tempo de execução como de compilação. O facto de ser baixo-nível, não só oferece o controlo total sobre as transformações realizadas no código, como também um excelente desempenho na sua realização. No entanto, a complexidade das transformações é muito elevada, exigindo que o utilizador possua algum conhecimento de *bytecode*. Com isto, existe um forte acoplamento dos conceitos implementados pela ferramenta sobre o *bytecode*,

sendo praticamente impossível a sua adaptação numa linguagem como o WASM. Devido à sua completude, o tamanho da ferramenta é relativamente grande.

No que toca ao Javassist, esta é uma ferramenta mais fácil de utilizar que o ASM, no entanto, muito mais limitada e ineficiente. Os conceitos existentes na ferramenta são lineares, contudo para realizar transformações mais complexas, estes tornam-se impraticáveis.

A ferramenta AspectJ oferece controlo quase total com uma complexidade muito reduzida, implementando os conceitos transversais das linguagens orientadas a aspetos. Esta é uma ferramenta bastante completa, permitindo realizar quase qualquer tipo de transformação no código Java.

Como se pode verificar, AspectJ é a única linguagem que fornece um elevado controlo sobre as modificações, com baixa complexidade e cujos conceitos são transversais (os conceitos podem servir de base para outras linguagens). Desta forma, conclui-se que a ferramenta WasmManipulator terá como base os conceitos aplicados nessa linguagem para melhor cumprir com os requisitos estabelecidos na Secção 4.1.

3.3. Linguagens para Procura e Substituição

Nesta secção são abordadas as tecnologias que foram consideradas para implementar a funcionalidade de *templates* (descrita na Secção 3.3 do Anexo B) na ferramenta WasmManipulator.

3.3.1. Xpath e XSLT

Xpath (XML Path Language) (MDN Contributors, XPath, 2021) e XSLT (Extensible Stylesheet Language Transformations) (MDN Contributors, XSLT, 2021) são ambas ferramentas que interagem entre si com o intuito de analisar e manipular conteúdo do tipo Extensible Markup Language (XML). Enquanto que o principal objetivo do Xpath (Team W. , XPath Tutorial, 2021) é procurar conteúdo num determinado código XML, o XSLT é uma linguagem que permite a transformação de um documento XML noutro documento XML (ou noutros formatos, como por exemplo HTML ou PDF), recorrendo internamente ao Xpath para identificar subconjuntos de elementos e processar as transformações.

Xpath converte cada campo presente no código XML para um nó específico, que possui um determinado contexto associado ao seu tipo (existem sete tipos de nós). Os vários nós encontram-se interligados entre si, através de *axis*, que representam a respetivas relações entre estes nós, como por exemplo, a relação "pai" e "filho", a relação de "descendentes", etc. Com recurso aos *axis*, é possível criar uma expressão conhecida por *Location Step* que permite a seleção de um ou mais nós do conteúdo XML. Estas expressões podem ser combinadas em *Location Paths* que basicamente são formados por um ou mais *Location Steps* que permitem a navegação pelos vários nós (Burke, 2001).

Ao contrário do conteúdo XML, a Abstract Syntax Tree (AST) associada ao código WASM é demasiado complexa para utilizar esta combinação de linguagens. Isto porque, a maior parte dos nós correspondem a instruções do código, uma vez que para além dos nós iniciais que representam secções do módulo, não há muito mais profundidade para além das

expressões utilizadas no corpo das funções. Como consequência a utilização de *Location Paths* seria demasiado extensa e muito pouco flexível.

3.3.2. CRAQL

Segundo (Johnson & Simha, 2019), CRAQL (Composable Repository Analysis and Query Language) é uma linguagem de pesquisa e consulta para código-fonte. Esta linguagem combina uma sintaxe de consulta declarativa do estilo SQL (Team W. , SQL Tutorial, 2021), com um componente estruturado e imperativo do tipo C (linguagem de programação C) para filtragem, e pré e pós-processamento de resultados. As *queries* podem ser compostas uma vez que o resultado de uma *query* pode ser utilizado como entrada para outra. Para tal, tanto o conteúdo de entrada, como o conteúdo resultante correspondem a conjuntos de ASTs. Estes resultados são combinados com um conjunto de informação adicional (*metadata*) sobre a AST obtida, que posteriormente poderá ser acedida pelo utilizador.

À semelhança do Xpath e XSLT, uma das razões pelo qual esta linguagem não foi adaptada para a ferramenta WasmManipulator está também relacionada com a AST produzida pelo código WASM. Neste caso, o problema deriva da complexidade e dimensão associadas à mesma, uma vez que para a maior parte dos casos, ao ser realizada uma determinada transformação, devido à natureza desta linguagem, seria necessário uma *query* muito mais extensa e elaborada do que o pretendido.

3.3.3. Stratego/XT

Com base em (Bravenboer, Kalleberg, Vermaas, & Visser, 2008), Stratego/XT é uma *framework* utilizada para o desenvolvimento de sistemas de transformação. Esta é composta por uma linguagem de transformação, o Stratego, e por uma coleção de ferramentas de transformação, o XT. Esta *framework* aborda todos os aspetos da construção de um sistema de manipulação de código, desde a especificação das transformações até à sua composição e aplicação.

O Stratego disponibiliza um conjunto de regras que permitem expressar transformações básicas. Este é baseado num controlo de estratégias que programam o modo como as regras são aplicadas. A linguagem também suporta regras dinâmicas, permitindo que sejam executadas de acordo com o ambiente onde são aplicadas, isto é, possibilitam a realização de transformações sensíveis ao contexto.

As ferramentas disponibilizadas pelo XT oferecem diversos recursos que poderão ser necessários nas infraestruturas de transformação, tais como, ferramentas para executar o *parse* ou gerar o resultado da transformação.

Apesar de ser uma *framework* bastante completa, que para além de permitir a composição e reutilização de transformações, também é composta uma biblioteca rica em transformações genéricas, esta não foi uma opção para a ferramenta WasmManipulator devido à sua complexidade e dimensão. Isto porque, para além de ser necessário a criação de um *parser* para a linguagem WASM, a declaração das transformações é feita de uma forma bastante ilegível e extensa.

3.3.4. Comby

Comby (Comby, 2021) é uma ferramenta utilizada para procurar e reescrever código. O facto desta ferramenta definir um *parser* de carácter geral, ideal para uma pesquisa eficiente em expressões formatadas com recurso a parênteses, fez como que fosse adaptada para integrar o WasmManipulator na implementação da funcionalidade de pesquisa por padrão (funcionalidade *templates* descrita na Secção 3.3 do Anexo B). Isto deve-se ao facto da linguagem WAT ser baseada em *S-expressions*, sendo por isso facilmente consultada e manipulada pelo Comby. Além disso, permite a identificação de variáveis, que é uma das funcionalidades requeridas para esta pesquisa.

Para o presente trabalho apenas é preciso conhecer o conceito de `hole`, que tem como objetivo armazenar uma determinada parte do código numa dada variável. Na Figura 31 encontra-se ilustrado um exemplo retirado da documentação (Comby, 2021), com a definição de um `hole` que permite o armazenamento do parâmetro passado à função `fmt.Println`. Neste caso qualquer que fosse o valor do parâmetro, ficaria armazenado na variável `arguments`.

```
fmt.Println(:[arguments])
```

Figura 31 - Código Go com exemplo de entrada do Comby (Comby, 2021)

O resultado da execução desta ferramenta não só nos fornece os blocos de código que coincidem com o padrão de entrada como também os valores associados a cada uma das variáveis. O modelo para o resultado encontra-se no formato JavaScript Object Notation (JSON). A Figura 32, baseada na documentação (Comby, 2021), apresenta o código JSON com o resultado da execução da ferramenta para o código definido na Figura 31 sobre o código apresentado na Figura 33. Depois de obtido, o modelo é tratado e convertido num modelo interno da ferramenta WasmManipulator.

```
{
  "uri": null,
  "matches": [{
      "range": {
        "start": { "offset": 0, "line": 1, "column": 1 },
        "end": { "offset": 21, "line": 1, "column": 22 }
      },
      "environment": [{
          "variable": "argument",
          "value": "\"hello!\"",
          "range": {
            "start": { "offset": 12, "line": 1, "column": 13 },
            "end": { "offset": 20, "line": 1, "column": 21 }
          }
      }],
      "matched": "fmt.Println(\"hello!\")"
    }
]}
```

Figura 32 - Código JSON com o resultado do exemplo do Comby (Comby, 2021)

```
fmt.Println("hello!")
```

Figura 33 - Código de entrada para procurar por padrão no Comby (Comby, 2021)

3.3.5. Análise Comparativa

Na Tabela 4 encontra-se um resumo da análise realizada para as diferentes ferramentas abordadas nesta secção.

Tabela 4 - Análise comparativa das ferramentas de pesquisa e substituição

	XPath/XSLT	***CRAQL***	***Stratego/XT***	***Comby***
Tipo de Pesquisa	Pesquisa por caminhos e expressões	Pesquisa por *query*	Pesquisa através de regras e estratégias	Pesquisa por padrão
Complexidade	Média	Média	Elevada	Baixa
Desempenho	Rápido	Rápido	Rápido	Rápido
Estabilidade	Estável	Não estável	Estável	Estável
Dificuldade na Integração	Impossível	Elevado	Elevado	Baixo

O XPath/XSLT é uma combinação de linguagens que permitem a manipulação de documentos XML baseando-se na definição de caminhos, onde cada elemento presente no caminho pode definir uma determinada expressão. O âmbito da linguagem afasta-se muito do que é pretendido para a funcionalidade dos *templates*, e por isso, não é possível a sua integração na ferramenta.

Em relação ao CRAQL, este é uma ferramenta de pesquisa e consulta de código-fonte, baseado na combinação de uma sintaxe declarativa do estilo SQL, com um componente estruturado e imperativo do tipo C (linguagem C). Para além de não ser estável, esta linguagem tem diversos problemas que levaram a não adaptação na ferramenta WasmManipulator, tais como, o facto de não ser de fácil integração, uma vez que teria de ser implementada do zero, ou de não ser muito compatível com a linguagem WASM.

O Stratego/XT consiste numa ferramenta que através da linguagem Stratego permite a pesquisa através de regras e estratégias. XT é apenas o conjunto de ferramentas disponibilizado para executar e realizar as manipulações. O Stratego foi descartado não só devido à complexidade que a sua integração acarreta, mas também devido à natureza da abordagem seguida pela linguagem, que por sua vez é também demasiado complexa para o pretendido.

Relativamente à ferramenta Comby, esta permite a pesquisa por padrões de código, incluindo a identificação de variáveis. Para além disso, é extremamente eficiente e compatível com o WAT, permitindo assim uma integração fácil com a ferramenta WasmManipulator. Sabendo isto, o Comby foi a ferramenta ideal para ser integrada na funcionalidade de *templates* na ferramenta WasmManipulator.

3.4. YAML

Por fim, será abordada a linguagem YAML Ain't Markup Language (YAML), que foi utilizada pela ferramenta para estruturar o código de transformação. Segundo a documentação (Net, 2021), YAML consiste numa linguagem de serialização de dados visualmente amigável e disponível para integração em qualquer linguagem de programação. Sabendo isto, utilizou-se esta linguagem na ferramenta desenvolvida neste trabalho para que as declarações de transformação fossem especificadas de forma estruturada, compacta e de fácil leitura. Com isto, não só a ferramenta é capaz de organizar e estruturar os vários elementos existentes na transformação para um modelo consistente, como também facilita a legibilidade e manutenção do código por parte do utilizador.

Os objetivos definidos na especificação da linguagem (Ben-Kiki, Evans, & Net, 2009) são:

- fácil leitura para humanos;
- corresponder às estruturas de dados nativas de linguagens ágeis;
- possuir um modelo consistente de forma a oferecer suporte a diversos tipos de ferramentas;
- portabilidade dos dados;
- suportar que seja processado cum uma passagem apenas;
- ser expressivo e extensível;
- e por fim, ser fácil de implementar e utilizar.

Para cumprir com os objetivos a linguagem teve de ser projetada de forma compacta e ao mesmo tempo composta por um vasto conjunto de elementos. No exemplo da Figura 34, baseado em (Erik, 2018), é possível verificar alguns desses elementos, que neste caso, são os necessários para o âmbito da linguagem utilizada no WasmManipulator. O código começa com três *hyphens* que indicam o início de um novo documento YAML. Uma vez que YAML suporta a definição de vários documentos dentro do mesmo ficheiro, estes *hyphens* são utilizados para que o *parser* reconheça o início de cada um. No caso das linguagens orientadas a objetos, cada documento é interpretado e armazenado numa instância do respetivo objeto. A definição das propriedades é feita com recurso a registos chave-valor, onde a chave corresponderá ao campo e o valor ao respetivo valor do campo.

```yaml
---
 doe: "a deer, a female deer"
 ray: "a drop of golden sun"
 pi: 3.14159
 xmas: true
 french-hens: 3
 calling-birds:
   - name: huey
     id: 1759
   - name: dewey
     id: 5689
   - name: louie
     id: 4192
   - name: fred
```

```
    id: 7498
xmas-fifth-day:
  calling-birds: four
  french-hens: 3
  golden-rings: 5
  partridges:
    count: 1
    location: >
     a pear tree
  turtle-doves: two
```

Figura 34 - Código YAML com exemplo da linguagem (Erik, 2018)

Na Figura 35 encontra-se ilustrado um modelo de dados conceptual associado ao código do exemplo, onde como se pode verificar, é composto por objetos, *arrays*, *strings*, booleanos e números. Neste modelo a definição de objetos é feita através de "`{}`" e de *arrays* é "`[]`".

```
{
  doe: string,
  ray: string,
  pi: number,
  xmas: boolean,
  french-hens: number,
  calling-birds: Array<{
    name: string,
    id: number,
  }>, // Array de objetos.
  xmas-fifth-day: {
    calling-birds: string,
    french-hens: number,
    golden-rings: number,
    partridges: {
      count: number,
      location: string,
    }, // Objeto.
    turtle-doves: string,
  }, // Objeto.
}
```

Figura 35 - Modelo de dados conceptual associado ao exemplo da linguagem YAML

CAPÍTULO 4.
ESPECIFICAÇÃO DA LINGUAGEM

As transformações para o código são definidas no WasmManipulator com recurso a uma linguagem específica estruturada em YAML. Estas transformações serão realizadas segundo um paradigma orientado a aspetos. Durante este processo, será também possível o acesso ao contexto do código, a criação de novos elementos, e a interpretação de expressões de forma estática ou em tempo de execução. Neste capítulo será abordada com detalhe a especificação desta linguagem, criada únicamente para ser usada na ferramenta WasmManipulator.

Na Secção 4.1 serão abordados os requisitos estabelecidos para a ferramenta, onde existirá uma breve descrição para cada um destes.

A linguagem é introduzida na Secção 4.2, onde são introduzidos os vários conceitos implementados na linguagem através de exemplos iterativos. Estes exemplos cobrem ainda todos os requisitos definidos na Secção 4.1.

A Secção 4.3 contém uma pequena explicação sobre como se deve proceder à execução da ferramenta, onde são descritas as opções de configuração disponibilizadas para essa execução.

Por fim, a Secção 4.4 exemplifica como deve ser integrado o programa gerado pela ferramenta após as transformações, quando engloba a utilização de um módulo JS auxiliar.

4.1. Requisitos

A ferramenta WasmManipulator tem como objetivo permitir a manipulação simples e flexível de código WASM. Com isto, a ferramenta procura oferecer ao utilizador a capacidade de aplicar diversos tipos de transformações, incluindo aquelas mais comuns nas linguagens orientadas a aspetos.

Sabendo isto, foram definidos os seguintes requisitos:

- **Adicionar transformação em vários pontos específicos do programa** - as modificações devem ser realizadas através de *advices* que poderão ser aplicados em mais do que um ponto do código. Estes pontos são chamados de *join-points*, e são recolhidos através de uma expressão conhecida por *pointcut*. Nestas expressões, o utilizador terá a capacidade de conjugar as diferentes funções de procura existentes na ferramenta.
- **Adicionar elementos ao programa** – para além de permitir que sejam aplicadas modificações ao código existente no módulo WASM, deve também permitir que sejam adicionados novos elementos ao contexto global. Estes elementos devem consistir em variáveis globais, funções e respetivos elementos das funções, e devem ser devidamente referenciados através de um identificador único e acessível no respetivo *scope* do ficheiro de transformações.
- **Aceder aos elementos criados no código** – através de um mecanismo de expressões, o utilizador deverá ter a possibilidade de aceder aos elementos criados no ficheiro de

transformação recorrendo ao seu identificador. Este acesso deve ter em conta o *scope* onde está a ser feita a definição e o acesso aos elementos.

- **Aceder ao contexto do ponto onde será incluída a transformação** - através do mesmo mecanismo de expressões, o utilizador deverá ter a possibilidade de utilizar os diferentes dados de contexto do ponto onde será incluído o código definido na transformação. Este contexto deve ser disponibilizado pelas funções declaradas na expressão do *pointcut*. Para além disto, também deve existir a possibilidade de definir novas variáveis locais para cada uma das funções onde as modificações são aplicadas.
- **Manipular dados disponibilizados na transformação** - utilizando o mesmo mecanismo de expressões, o utilizador deve ter a capacidade de modificar os dados referenciados através de funções de transformação.
- **Executar transformações sensíveis ao contexto da execução** - através de um novo mecanismo de expressões, deve ser possível realizar transformações sensíveis ao contexto em tempo de execução. Para aumentar a capacidade das transformações, estas expressões devem permitir um funcionamento semelhante ao código JS.
- **Permitir a utilização de tipos complexos** - para além dos tipos existentes na especificação do WASM, devem ser acrescentados os tipos de dados *string*, *array* e mapa. Para cada um destes tipos deve existir a possibilidade de aceder às respectivas propriedades e métodos JS (exemplo, para obter o tamanho de um *array* deve ser feito o acesso à propriedade *length* – "*array.length*"). Este acesso deve ser feito no mesmo mecanismo de expressões interpretado em tempo de execução.
- **Executar sem a interação do utilizador** - isto não significa que a ferramenta irá fazer todo o processo de transformação por si só. O que significa é que, após terem sido definidas as transformações pelo utilizador, e configurado o ambiente de execução, a ferramenta possuirá um funcionamento autónomo, isto é, ao ler o ficheiro com o código WASM e o ficheiro com as respetivas transformações, será capaz de produzir o resultado esperado sem qualquer interação do utilizador.

4.2. Linguagem para transformação

Nesta secção vai ser apresentado um exemplo construído de forma iterativa onde progressivamente serão aplicadas determinadas transformações que permitirão introduzir as diversas funcionalidades disponibilizadas pela ferramenta e como estas devem ser definidas na linguagem de transformação.

4.2.1. Exemplo Inicial

O exemplo consiste num programa WASM que permite realizar determinadas operações matemáticas. A Figura 36 ilustra o código do programa que, inicialmente, apenas permite a soma de dois inteiros. Para isso foi criada a função exportada `$math` que internamente invoca a função `$add`, onde é calculada a soma dos dois parâmetros recebidos e devolvido o resultado (inteiros *32-bits*).

```
(module
    (func $math (param $lv i32) (param $rv i32) (result i32)
        (call $add (local.get $lv) (local.get $rv))
    )
    (func $add (param $lv i32) (param $rv i32) (result i32)
        (i32.add (local.get $lv) (local.get $rv))
    )
    (export "math" (func $math))
)
```

Figura 36 - Código WAT inicial para o exemplo da especificação da linguagem

Pretende-se agora modificar este código de forma a garantir que todas as chamadas às funções são registadas. Isto será feito garantindo que antes de todas as chamadas a qualquer função, é feita uma chamada a uma função `before_call` e que, de forma idêntica, é chamada uma função `after_call` após o retorno de qualquer função.

As funções `after_call` e `before_call` estão, por conveniência, definidas num módulo distinto com o nome `env`, e poderão realizar qualquer atividade relevante (contagem de chamadas, registo de informação, etc) que o utilizador pretenda.

Estando criadas as funções `before_call` ou `after_call` de acordo com as necessidades, irá agora ser definida uma transformação que irá garantir que o código original é modificado de forma a que essas funções sejam chamadas na altura devida.

Como as transformações na ferramenta são baseadas no paradigma orientado a aspetos, a unidade básica de transformação consiste na definição de um *advice*. Desta forma, foi criado o *advice* `instrument_call` que permite a instrumentação do código, registando cada chamada realizada. Esta instrumentação é composta por uma chamada à função `before_call` antes da instrução de chamada, e uma chamada à função `after_call` após a mesma instrução. Na Figura 37 está representado o código necessário à transformação.

```
aspects:
  context:
    functions:
      before_call:
        imported:
          module: env
          field: bef_call
      after_call:
        imported:
          module: env
          field: aft_call
  advices:
    instrument_call:
      pointcut: () => call(* *(..))
      advice: >
        (call %before_call%)
        %this%
        (call %after_call%)
```

Figura 37 – Código com a transformação para instrumentar exemplo da especificação da linguagem

A definição do aspeto inicia-se com a palavra aspects. Este aspeto está subdividido em duas seções: `context` e `advices`. A seção context contém informação que se pretende inserir no contexto global, tal como novas variáveis ou então, como é aqui o caso, novas funções.

Aqui são criadas as funções `before_call` e `after_call`, incluídas no contexto global dos aspetos (ou seja, podem ser acedidas por qualquer código definido na secção `aspects`). Ambas as funções são importadas a partir do módulo `env`, nos campos `bef_call` e `aft_call` respetivamente.

Enquanto que a secção `context` descreve simples acréscimos aquilo que já existe no programa original (mais variáveis, mais funções), a secção `advices` descreve modificações ao código existente. Estas modificações são descritas através de pares de *pointcuts* (que identificam localizações específicas no código) e *advices* (que descrevem as modificações a aplicar nessas localizações).

Para modificar as instruções de chamada foi definido o *pointcut* `call`. Este foi configurado para aceitar qualquer tipo de chamada através da configuração `* *(..)`, uma que a sintaxe para a configuração é `<return> <name>(<params>)`. Por outras palavras, este pointcut identifica todas as localizações do código original nas quais é realizada uma chamada a uma função. Cada uma das localizações identificadas por um pointcut é designada por joinpoint na nomenclatura de orientação a aspetos. Neste caso, o pointcut irá identificar um único joinpoint, conforme ilustrado na Figura 38.

```
(module
    (func $math (param $lv i32) (param $rv i32) (result i32)
        (call $add (local.get $lv) (local.get $rv))
    )
    (func $add (param $lv i32) (param $rv i32) (result i32)
        (i32.add (local.get $lv) (local.get $rv))
    )
    (export "math" (func $math))
)
```

Figura 38 - Código WAT inicial para o exemplo da especificação da linguagem com o *join-point* assinalado

O código para o *advice* descreve a modificação que irá ser aplicada em cada *join-point*. O advice irá sempre substituir o código do *join-point* (em termos de orientação a aspetos atua sempre como uma operação *around*). No entanto, é possível aceder e reinserir o código do *join-point* dentro do *advice* com recurso à variável `this`. Esta variável está presente em todos os advices definidos na transformação, assumindo sempre o valor das respetivas instruções associadas ao *join-point* que se encontra a ser executado. Se o *advice* reinserir estas instruções, o código original do pointcut é preservado no código transformado, o que permite implementar *advices* que são, em termos práticos, executados antes ou depois do joinpoint (usualmente designados por advices *before* e *after*). Isto pode ser observado no exemplo do advice mostrado na Figura 37, onde, a variável `this` foi inserida entre ambas as chamadas às funções de instrumentação.

A utilização de variáveis no código engloba sempre a utilização de determinadas expressões específicas à ferramenta. Neste caso, foram utilizadas as *static expressions* (ver Secção 3.1) que por definição permitem o acesso e manipulação de variáveis de transformação no código. Estas expressões são delimitadas pelo caractere `%`, havendo assim três expressões definidas no programa, o `%this%` que permitirá a inserção das instruções da chamada presente no *join-point*, o `%before_call%` e o `%after_after%` que permitem a inserção do respetivo índice associado ao elemento das funções de instrumentação adicionadas na transformação.

O resultado da transformação encontra-se ilustrado na Figura 39.

```
(module
  (type $t0 (func (param i32 i32) (result i32)))
  (type $t1 (func))
  (import "env" "bef_call" (func $env.before_call (type $t1)))
  (import "env" "aft_call" (func $env.after_call (type $t1)))
  (func $math (type $t0) (param $p0 i32) (param $p1 i32) (result i32)
    (call $env.before_call)
    (call $f3
      (local.get $p0)
      (local.get $p1))
    (call $env.after_call))
  (func $f3 (type $t0) (param $p0 i32) (param $p1 i32) (result i32)
    (i32.add
      (local.get $p0)
      (local.get $p1)))
  (export "math" (func $math)))
```

Figura 39 - Código WAT resultante da transformação para instrumentar exemplo da especificação da linguagem

Isto finaliza o primeiro exemplo, onde foram exemplificadas as seguintes funcionalidades:

- Criar um *advice* definindo um *pointcut* e o respetivo código de transformação.
- Criar novas funções.
- Importar funções criadas.
- *Static Expressions* no acesso a variáveis.

4.2.2. Modo Inteligente

Como é possível verificar na Figura 39, a função `$math` após a transformação realizada acima já não está correta, uma vez que já não devolve o resultado da adição. Isto poderia ser deliberado (o objetivo da transformação poderia implicar a alteração do valor de retorno da função), mas tal não é na verdade o caso. Isto deve-se ao facto da instrução com a chamada à função `$add` não ser a última chamada do código, uma vez que foi inserida a chamada à função `after` após essa instrução. Para corrigir este tipo de problemas, seria possível redesenhar o *advice* para preservar a devolução do valor original. No entanto, para simplificar este procedimento, foi criado o modo `smart` (ver Secção 4 do Anexo B), que permitiu que cada *advice* preserve o valor de retorno das instruções dos *join-points*, de forma inteligente, procedendo às alterações necessárias para que a instrução de retorno se

mantenha. Para tornar o problema ainda mais visível, foi adicionado ao código do módulo original a operação matemática de subtração (Figura 40). Ao utilizar o modo `smart` (Figura 41), o código resultante da transformação que se encontra ilustrado na Figura 42, independentemente do ponto de retorno da função `$math`, devolve sempre o valor do resultado da operação.

Para ilustrar este conceito, vamos considerar a função definida na Figura 40. Esta função devolve o resultado da soma ou subtração de dois dos seus argumentos, em função do valor de um terceiro argumento.

Pretende-se agora implementar uma transformação similar àquela definida na Secção 4.2.1, garantindo, no entanto, que o valor devolvido não é alterado. Para o conseguir será suficiente indicar o advice como sendo `smart`.

Ao utilizar o modo *smart* (Figura 41), o código resultante da transformação que se encontra ilustrado na Figura 42, independentemente do ponto de retorno da função $math, devolve sempre o valor do resultado da operação.

```
(module
    (func $math (param $code i32) (param $lv i32) (param $rv i32) (result i32)
        (block $B0
            (br_if $B0 (i32.eq (local.get $code) (i32.const 0)))
            (return (call $add (local.get $lv) (local.get $rv)))
        )
        (call $sub (local.get $lv) (local.get $rv))
    )
    (func $add (param $lv i32) (param $rv i32) (result i32)
        (i32.add (local.get $lv) (local.get $rv))
    )
    (func $sub (param $lv i32) (param $rv i32) (result i32)
        (i32.sub (local.get $lv) (local.get $rv))
    )
    (export "math" (func $math))
)
```

Figura 40 - Código WAT atualizado com a inserção da operação de subtração no exemplo da especificação da linguagem

```
aspects:
  ...
  advices:
    instrument_call:
      pointcut: () => call(* *(..))
      smart: true
      advice: >
        ...
```

Figura 41 - Código com a transformação inteligente para corrigir a instrumentação adicionada ao exemplo da especificação da linguagem

```
(module
  (type $t0 (func (param i32 i32 i32) (result i32)))
  (type $t1 (func (param i32 i32) (result i32)))
  (type $t2 (func))
  (import "env" "bef_call" (func $env.before_call (type $t2)))
  (import "env" "aft_call" (func $env.after_call (type $t2)))
  (func $math (type $t0) (param $p0 i32) (param $p1 i32) (param $p2 i32) (resul
t i32)
    (local $l3 i32) (local $l4 i32)
    (block $B0
      (br_if $B0
        (i32.eq
          (local.get $p0)
          (i32.const 0)))
      (call $env.before_call)
      (local.set $l3
        (call $f3
          (local.get $p1)
          (local.get $p2)))
      (call $env.after_call)
      (return
        (local.get $l3)))
    (call $env.before_call)
    (local.set $l4
      (call $f4
        (local.get $p1)
        (local.get $p2)))
    (call $env.after_call)
    (local.get $l4))
  (func $f3 (type $t1) (param $p0 i32) (param $p1 i32) (result i32)
    (i32.add
      (local.get $p0)
      (local.get $p1)))
  (func $f4 (type $t1) (param $p0 i32) (param $p1 i32) (result i32)
    (i32.sub
      (local.get $p0)
      (local.get $p1)))
  (export "math" (func $math)))
```

Figura 42 – Código WAT resultante da transformação inteligente para instrumentar exemplo da especificação da linguagem

Nesta secção foi demonstrada a aplicação do modo *smart*, que permite preservar o valor de retorno no código alterado para uma transformação.

4.2.3. Adicionar contexto à instrumentação

Na transformação da Figura 43 foram adicionados dois argumentos às funções de instrumentação de forma a que seja possível identificar a função que invocou e a que foi chamada. O código do *advice* foi também alterado para incluir a identificação das funções nos argumentos da instrução, sendo este o índice numérico da função. Para obter este índice foi necessário aceder ao campo `order` da respetiva função no contexto do *pointcut* `call`.

```
aspects:
  context:
    functions:
      before_call:
        args:
          - name: from
            type: i32
          - name: to
            type: i32
        imported:
          module: env
          field: bef_call
      after_call:
        args:
          - name: from
            type: i32
          - name: to
            type: i32
        imported:
          module: env
          field: aft_call
  advices:
    instrument_call:
      pointcut: () => call(* *(..))
      smart: true
      advice: >
        (call %before_call% (i32.const %call.caller.order%) (i32.const %call.ca
llee.order%))
        %this%
        (call %after_call% (i32.const %call.caller.order%) (i32.const %call.cal
lee.order%))
```

Figura 43 – Código com a transformação para inserir quais as funções que participam na instrumentação adicionada ao exemplo da especificação da linguagem

A transformação demonstrada nesta secção permitiu apresentar de forma breve o acesso aos dados de contexto fornecidos pelas funções de *pointcut* utilizadas na expressão. Para além disso, foi demonstrado a forma como são definidos os parâmetros nas funções inseridas.

4.2.4. Reverter Argumentos das Chamadas a Funções

Neste secção é demonstrada uma transformação que efetua a reversão dos argumentos das chamadas realizadas no módulo. Para isso, foi criado um *advice*, o `reverse_call_args`, que pretende modificar instruções referentes a chamadas a funções. O código do *advice*está ilustrado na Figura 44.

```
pointcuts:
  any_call: () => call(* *(..))
aspects:
  context:
    functions:
      before_call:
        args:
          - name: from
            type: i32
```

```
          - name: to
            type: i32
        imported:
          module: env
          field: bef_call
      after_call:
        args:
          - name: from
            type: i32
          - name: to
            type: i32
        imported:
          module: env
          field: aft_call
  advices:
    reverse_call_args:
      pointcut: () => any_call()
      advice: >
        (call %call.callee.index% %call.args:map((v) => v.instr):reverse():stri
ng()%)
    instrument_call:
      pointcut: () => any_call()
      smart: true
      advice: >
        (call %before_call% (i32.const %call.caller.order%) (i32.const %call.ca
llee.order%))
        %this%
        (call %after_call% (i32.const %call.caller.order%) (i32.const %call.cal
lee.order%))
```

Figura 44 - Código com a transformação para reverter a ordem dos operadores no exemplo da especificação da linguagem

Uma vez que a expressão para os *pointcuts* é igual para ambos os *advices*, esta foi encapsulada num *pointcut* global (ver Secção 1.6 do Anexo B) que passou a ser invocado no *pointcut* de ambos os *advices*.

O código para o novo *advice* é formado por duas *static expressions*. A primeira insere o índice da função invocada no código da nova chamada. A segunda tem como função inserir os argumentos da chamada mas por ordem inversa. Esta última introduz o conceito de manipulação estática de dados através de funções de transformação pré-implementadas na ferramenta (sintaxe `:<method>(<arguments>)`). Segundo um determinado valor de entrada, estas funções geram sempre um valor de saída, podendo este ser configurado de acordo com o tipo de função, como por exemplo, a função `:map()` recebe na sua configuração um argumento do tipo *lambda* (MDN Contributors, Arrow function expressions, 2021) que define o valor de saída para cada um dos elementos de entrada.

A transformação demonstrada nesta secção apresentou o conceito de *pointcut* global, que permite a reutilização de expressões de *pointcut* por vários *advices*, e ainda a utilização de algumas das funções de manipulação de dados disponibilizadas nas *static expressions*.

4.2.5. Ordenar Aplicação das Transformações

O código da transformação ilustrado na Figura 44 é suscetível a problemas relacionados com a ordem de aplicação dos *advices*. Apesar do *advice* `reverse_call_args` ter sido definido antes do *advice* `instrument_call`, este poderá não ser executado primeiro, pois a ordem pelo qual os *advices* são executados não depende da ordem pelo qual que estão definidos, e por isso, nem sempre é a mesma. Desta forma, quando o *advice* `reverse_call_args` é executado depois do `instrument_call`, a transformação para além de ser aplicada às chamadas com as operações, poderá ser também aplicada às funções de instrumentação. Esta é uma questão transversal a todas as tecnologias orientadas a asptos: o resultado da transformação pode depender da ordem de aplicação dos advices. Desta forma, foi incluído o campo `order` nos respetivos *advices* para que a ordem pelo qual são executados seja sempre a mesma. O código para os *advices* está ilustrado na Figura 45.

```
...
aspects:
  ...
  advices:
    instrument_call:
      pointcut: () => any_call()
      smart: true
      order: 1
      advice: >
        (call %before_call% (i32.const %call.caller.order%) (i32.const %call.ca
llee.order%))
        %this%
        (call %after_call% (i32.const %call.caller.order%) (i32.const %call.cal
lee.order%))
    reverse_call_args:
      pointcut: () => any_call()
      order: 0
      advice: >
        (call %call.callee.index% %call.args:map((v) => v.instr):reverse():stri
ng()%)
```

Figura 45 - Código com a transformação ordenada que corrige a transformação que reverte a ordem dos operadores no exemplo da especificação da linguagem

Nesta secção foi demonstrada a importância da ordem pelo qual os *advice* são aplicados, sendo introduzido o campo `order` nas transformações.

4.2.6. Adicionar Multiplicação

Na Figura 46 encontra-se ilustrado o código da transformação que permitiu adicionar à função `$math` a operação matemática de multiplicação. Para isso foi criado o *advice* `add_mul_before` que passou a ser o primeiro *advice* a executar na transformação.

```
...
aspects:
  context:
    functions:
      ...
      mul:
        args:
          - name: value
            type: i32
          - name: times
            type: i32
        result: i32
        code: >
          (i32.mul (local.get %value%) (local.get %times%))
  advices:
    instrument_call:
      pointcut: () => any_call()
      smart: true
      order: 2
      advice: >
        (call %before_call% (i32.const %call.caller.order%) (i32.const %call.ca
llee.order%))
        %this%
        (call %after_call% (i32.const %call.caller.order%) (i32.const %call.cal
lee.order%))
    reverse_call_args:
      pointcut: () => any_call()
      order: 1
      advice: >
        (call %call.callee.index% %call.args:map((v) => v.instr):reverse():stri
ng()%)
    add_mul_before:
      pointcut: () => func(i32 *(i32 %code%, i32 %lv%, i32 %rv%), exported)
      order: 0
      advice: >
        (block $B0
          (br_if $B0 (i32.ne (local.get %code.name%) (i32.const 1)))
          (return (call %mul% (local.get %lv.name%) (local.get %rv.name%)))
        )
        %this%
```

Figura 46 - Código com a transformação para inserir a operação de multiplicação no exemplo da especificação da linguagem

Foi implementada uma nova função interna, a `mul`, que calcula o resultado da multiplicação de dois inteiros (*32-bit*) passados nos argumentos. A chamada a esta função foi inserida no início da função `$math`, através do *advice* `add_mul_before`. Para que as alterações aplicadas nos outros *advices* incluíssem esta chamada, foi necessário adaptar a ordem de execução para que este fosse a primeira transformação a ser aplicada.

O código para o *advice* `add_mul_before` utiliza os contexto definido na configuração do *pointcut* `func` para implementar esta alteração. Este *pointcut* está configurado para obter apenas funções exportadas definidas com três parâmetros do tipo inteiro (*32-bit*) e um valor

de retorno do tipo inteiro (*32-bit*). Nesta definição foram declaradas as variáveis `code`, `lv`, `rv` que correspondem a cada um dos respetivos parâmetros da função.

O acesso aos parâmetros da função também poderia ser feito através dos parâmetros de contexto do `pointcut`. Este contexto está associado à função onde está incluída a execução do *join-point*, logo, para além do acesso aos parâmetros da função, também é possível aceder às variáveis locais da mesma (`local` ao invés de `param`). Na Figura 47 está ilustrado o código com a utilização deste contexto ao invés do contexto fornecido pela configuração do *pointcut* `func`.

```
...
aspects:
  ...
  advices:
    ...
    add_mul_before:
      pointcut: (i32.param[0] code, i32.param[1] lv, i32.param[2] rv) => func(i
32 *(i32, i32, i32), exported)
      order: 0
      advice: >
        (block $B0
          (br_if $B0 (i32.ne (local.get %code%) (i32.const 1)))
          (return (call %mul% (local.get %lv%) (local.get %rv%)))
        )
        %this%
```

Figura 47 - Código com a transformação onde é feito o acesso ao contexto do *pointuct* inserir a operação de multiplicação no exemplo da especificação da linguagem

Na transformação da Figura 46 foi demonstrado o acesso ao contexto definido na configuração dos *pointcuts* para utilizar no código da transformação os respetivos parâmetros da função onde está a ser aplicada a alteração. Para a transformação da Figura 47, foi realizada a mesma alteração, no entanto, a transformação invés de englobar o acesso ao contexto definido nos *pointcuts*, utiliza os parâmetros da função através do acesso ao contexto declarado nos parâmetros da definição do *pointcut*.

4.2.7. Instrumentar Operações

A próxima transformação consiste no registo das várias operações que são feitas internamente no código. Para isso foi criado o *advice* `track_operations`, que para cada operação invoca uma função específica, passando-lhe o código da operação e o respetivo resultado. O código da transformação está ilustrado na Figura 48.

```
...
templates:
  operation: (i32.%name% %first% %second%)
aspects:
  context:
    functions:
      log_operation:
        args:
          - name: code
```

```
            type: i32
          - name: result
            type: i32
        imported:
          module: env
          field: log_operation
      ...
  advices:
    ...
    track_operations:
      pointcut: () => func(* *(..), internal) && template(operation)
      all: true
      smart: true
      variables:
        res: i32 = 0
        code: i32 = 0
      advice: >
        (local.set %code%
          (i32.const
            %"-1":assert(() => operation:select(name) != "add"
              && operation:select(name) != "sub"
              && operation:select(name) != "mul")%
            %"0":assert(() => operation:select(name) == "add")%
            %"1":assert(() => operation:select(name) == "sub")%
            %"2":assert(() => operation:select(name) == "mul")%
          )
        )
        (target (local.tee %res% (i32.%operation:select(name)% %operation:selec
t(first)% %operation:select(second)%)))
        (call %log_operation% (local.get %code%) (local.get %res%))
```

Figura 48 - Código com a transformação onde é feito o registo das operações executadas no exemplo da especificação da linguagem

Para obter todas as operações internas, foi definida uma expressão no *pointcut* que combinou a função `func` com a função `template` através do operador lógico "*AND*" (`&&`). Com a função de *pointcut* `func` configurada com o valor `internal`, limitou-se a pesquisa apenas às funções internas do programa, isto é, que nem são importadas nem exportadas. Depois com a função `template`, foi possível realizar uma pesquisa por padrão utilizando o *template* `operation` definido no objeto `templates`.

O *template* `operation` define uma instrução para inteiros que recebe dois argumentos. O nome e os argumentos podem assumir qualquer valor, sendo armazenados na respetiva variável (`name`, `first` e `second`). Estas variáveis são depois acedidas no código do *advice* através da função `:select()`, invocada sobre o identificador do *template* (`operation`).

No *advice* foram definidas duas variáveis auxiliares, `res` e `code`. A variável `res` serviu para armazenar o resultado da execução da operação, que para além de ser retornado pela alteração através da instrução `target` (ver Secção 4), foi também utilizado na chamada à função de registo que ocorreu após este cálculo. A variável `code` armazena o código da operação e foi utilizada na chamada à função de registo. Esta é inicializada com recurso a uma constante cujo valor depende do nome da operação. Para implementar a condição foi utilizada a função de transformação `assert` que é configurada com uma `lambda` que devolve

um booleano, que caso seja verdadeiro, continua a cadeia de transformações devolvendo o valor de entrada, e caso não seja, esta imprime uma *string* vazia, interrompendo de imediato a cadeia de transformações que a sucede.

O código da transformação também inclui a definição da função de registo (`log_operation`), cuja implementação deve ser feita no lado do hospedeiro uma vez que é importada (campo `log_operation` do módulo `env`). Esta recebe como argumentos o código da operação e o respetivo resultado (inteiros *32-bit*).

Neste caso, não foi necessário adicionar o campo `order` para indicar que o *advice* deveria ser executado por último, uma vez que por predefinição os *advices* com o campo `order` especificado têm prioridade.

Por último, como o *advice* deveria ser executado para todas as funções, incluindo aquelas adicionadas no código da transformação, foi necessário ativar o campo `all`. Quando este não está ativo, o *advice* é aplicado apenas às funções presentes no código original do módulo.

Nesta secção foram demonstradas as seguintes funcionalidades:

- Combinação de funções de *pointcut* na expressão.
- Utilização do `target` para definir instrução de retorno no modo *smart*.
- Pesquisa por padrão através dos *templates*.
- Utilizar variáveis locais no *advice*.
- Incluir funções adicionadas nas transformações do *advice* através do campo `all`.

4.2.8. Restringir Operações Instrumentadas

Para evitar que a transformação realizada na Secção 4.2.7 seja aplicada a outras operações para além das conhecidas, foi alterada a forma como o *template* estava definido para restringir a pesquisa às operações `add`, `sub` e `mul`. Para isso recorreu-se às *template keywords* (ver Secção 3.3.1 do Anexo B), onde se utilizou a função `:include_one()` com os *templates* `op_add`, `op_sub` e `op_mul`, para restringir o nome das operações para coincidir com uma destas operações. Para além disso, apenas para efeitos demonstrativos, a instrução foi isolada no *template* `op_i32`, e no *template* base foi declarada a variável `name` através da função `:defines()`, ditando assim que esta deve ser definida no *template* `op_i32`. O código está ilustrado na Figura 49.

```
...
templates:
  operation: (%instr:includes(op_i32):defines(name)% %first% %second%)
  op_i32: "i32.%name:includes_one(op_add, op_sub, op_mul)%"
  op_add: "add"
  op_sub: "sub"
  op_mul: "mul"
aspects:
  ...
  advices:
    ...
    track_operations:
      pointcut: () => func(* *(..), internal) && template(operation)
```

```
      all: true
      smart: true
      variables:
        res: i32 = 0
        code: i32 = 0
      advice: >
        (local.set %code%
          (i32.const
            %"0":assert(() => operation:select(name) == "add")%
            %"1":assert(() => operation:select(name) == "sub")%
            %"2":assert(() => operation:select(name) == "mul")%
          )
        )
        (target (local.tee %res% (i32.%operation:select(name)% %operation:selec
t(first)% %operation:select(second)%)))
        (call %log_operation% (local.get %code%) (local.get %res%))
```

Figura 49 - Código com a transformação onde é feito a combinação de *templates* para proteger as operações a que o registo das operações é feito no exemplo da especificação da linguagem

Esta transformação permitiu demonstrar a utilização das *template keywords* dentro dos *templates*.

4.2.9. Contagem da Execução de Operações

Seguiu-se uma transformação, ilustrada na Figura 50, que visa inserir no programa a contagem do número de vezes que são efetuadas operações. Esta transformação foi aplicada no *advice* `track_operations` (criado na Secção 4.2.7), e engloba a criação da variável global `operations_count` que é incrementada sempre que uma operação é executada.

```
...
aspects:
  start: >
    (global.set %operations_count% (call %initial_operations_count%))
  context:
    functions:
      ...
      initial_operations_count:
        result: i32
        imported:
          module: env
          field: initial_operations_count
      get_operations_count:
        result: i32
        exported: get_operations_count
        code: >
          (global.get %operations_count%)
    variables:
      operations_count: i32 = 0
  advices:
    ...
    track_operations:
      pointcut: () => func(* *(..), internal) && template(operation)
      all: true
      smart: true
      variables:
        res: i32 = 0
```

```
        code: i32 = 0
      advice: >
        (global.set %operations_count% (global.get %operations_count%))
        (local.set %code%
            %"0":assert(() => operation:select(name) == "add")%
            %"1":assert(() => operation:select(name) == "sub")%
            %"2":assert(() => operation:select(name) == "mul")%
          )
        )
        (target (local.tee %res% (i32.%operation:select(name)% %operation:selec
t(first)% %operation:select(second)%)))
        (call %log_operation% (local.get %code%) (local.get %res%))
```

Figura 50 - Código com a transformação para realizar a contagem do número de vezes que uma operação é executada no exemplo da especificação da linguagem

O valor inicial da variável `operations_count` é configurado pelo hospedeiro através da função importada `initial_operations_count` (campo `initial_operations_count` do módulo `env`) que é invocada no início da execução do módulo, dentro da função inicial do módulo. O respetivo código é inserido através da definição no objeto `start`, que permite que seja inserido código no final de uma função inicial existente no módulo, ou caso não exista, que esta seja criada.

Foi também criada uma nova função, exportada para o hospedeiro com o nome `get_operations_count`, que permite devolver o valor da contagem.

Nesta secção foram demonstradas as seguintes funcionalidades:

- Exportar funções.
- Função inicial.
- Variáveis globais.

4.2.10. Instrumentação com Informação Textual

Até ao momento, tanto nas funções de instrumentação como na função de registo das operações tem utilizado valores do tipo inteiro (*32-bit*) para referenciar as respetivas funções e operações. Estes valores são passados para o lado do hospedeiro, englobando não só que este tenha conhecimento de aspetos que teoricamente deveriam ser internos ao módulo, que são a ordem pelo qual as funções estão dispostas no módulo, como também que esteja sempre sincronizado com os código das operações definidos no módulo. Para evitar isto, estas referências passaram a ser do tipo *string*. Desta forma, os índices das funções foram trocados pelos respetivos nomes, e os códigos das operações pelo respetivo nome da operação. Esta transformação encontra-se ilustrada na Figura 51.

```
...
aspects:
  ...
  context:
    functions:
      before_call:
        args:
          - name: from
            type: string
          - name: to
            type: string
        imported:
          module: env
          field: bef_call
      after_call:
        args:
          - name: from
            type: string
          - name: to
            type: string
        imported:
          module: env
          field: aft_call
      log_operation:
        args:
          - name: name
            type: string
          - name: result
            type: i32
        imported:
          module: env
          field: log_operation
      …
    ...
  advices:
    instrument_call:
      pointcut: () => any_call()
      smart: true
      all: true
      order: 2
      advice: >
        (call %before_call% /%'"';call.caller.name;'"'%/ /%'"';call.callee.name
;'"'%/)
        %this%
        (call %after_call% /%'"';call.caller.name;'"'%/ /%'"';call.callee.name;
'"'%/)
    reverse_call_args:
      pointcut: () => any_call()
      all: true
      order: 1
      advice: >
        (call %call.callee.index% %call.args:map((v) => v.instr):reverse():stri
ng()%)
    add_mul_before:
      pointcut: (i32.param[0] code, i32.param[1] lv, i32.param[2] rv) => func(i
32 *(i32, i32, i32), exported)
      order: 0
      advice: >
```

```
        (block $B0
          (br_if $B0 (i32.ne (local.get %code%) (i32.const 1)))
          (return (call %mul% (local.get %lv%) (local.get %rv%)))
        )
        %this%
    track_operations:
      pointcut: () => func(* *(..), internal) && template(operation)
      all: true
      smart: true
      variables:
        res: i32 = 0
      advice: >
        (global.set %operations_count% (global.get %operations_count%))
        (target (local.tee %res% (i32.%operation:select(name)% %operation:selec
t(first)% %operation:select(second)%)))
        (call %log_operation% /%'"';operation:select(name);'"'%/ (local.get %re
s%))
```

Figura 51 - Código com a transformação onde são aplicadas *strings* nas transformações de instrumentação aplicadas ao exemplo da especificação da linguagem

O tipo *string* não existe na especificação da linguagem WASM e por isso engloba transformações extra por parte da ferramenta. Estas transformações são realizadas com recurso ao JS, pelo que para além do módulo WASM gerado, é gerado um módulo JS que passará a ser o ponto de entrada para a execução correta do programa WASM. Para além do tipo *string*, também existem os tipos *array* e mapa (ver Secção 1.3 do Anexo B).

Para utilizar estes tipos adicionais é necessário recorrer às *runtime expressions*, que consistem em expressões interpretadas e executadas em tempo de execução, com recurso à função `eval` do JS (ver na Secção 3.2). Estas expressões são delimitadas por barras (`/`), e tal como os tipos, englobam transformações extras ao módulo, e dependem do JS para executar. No código da transformação, estas estão a ser utilizadas como argumentos do tipo *string* para as funções de instrumentação e registo de operações. Por exemplo, a expressão `/%'"';operation:select(name);'"'%/` consiste numa *runtime expressions* que possui internamente a definição de uma *static expression*. Desta forma, como a *static expression* é executada estaticamente, no caso da operação ser do tipo `add`, o resultado será `/"add"/`, e como consequência, esta expressão consistirá numa *string* com o nome da operação que posteriormente será passada para o ambiente hospedeiro.

Nesta secção foram introduzidos os contextos *runtime* das transformações, que exigem a utilização de um módulo JS auxiliar. Esta transformação demonstrou a utilização de tipos adicionais, neste caso *string*, e a utilização básica das r*untime expressions*.

4.2.11. Histórico da Execução de Operações

Na transformação da Figura 52 foi implementada uma melhoria na contagem das operações executadas para que passe a guardar o histórico de execução.

```
...
aspects:
  context:
    functions:
      ...
      initial_operations_count:
        args:
          - name: value
            type: i32
        exported: set_initial_operations_count
        code: >
          (global.set #operations_history /[]/)
          (block $B0
            (br_if $B0
              (i32.lt_s
                (local.tee %value% /#value - 1/)
                (i32.const 0)
              )
            )
            (loop $L0
              (local.set #value /#value/)
              (global.set #operations_history[value] /null/)
              (br_if $L0
                (i32.gt_s
                  (local.tee %value% /#value - 1/)
                  (i32.const -1)
                )
              )
            )
          )
      get_operations_count:
        result: i32
        exported: get_operations_count
        code: >
          /#operations_history.length/
      get_operations_history:
        result: "[]i32"
        exported: get_operations_history
        code: >
          #operations_history
      push_operation:
        args:
          - name: name
            type: string
        variables:
          index: i32 = 0
        code: >
          (local.set #index /#operations_history.length/)
          (local.set #index /#index/) (; register index on JS :)
          (global.set #operations_history[index] /#name/)
    variables:
      operations_history: "[]string = []"
  advices:
    ...
    track_operations:
      pointcut: () => func(* *(..), internal) && template(operation)
      all: true
      smart: true
```

```
      variables:
        res: i32 = 0
      advice: >
        (call %push_operation% /%'"';operation:select(name);'"'%/)
        (target (local.tee %res% (i32.%operation:select(name)% %operation:selec
t(first)% %operation:select(second)%)))
        (call %log_operation% /%'"';operation:select(name);'"'%/ (local.get %re
s%))
```

Figura 52 - Código com a transformação onde é guardado o histórico da execução das operações no exemplo da especificação da linguagem

Primeiro alterou-se a variável que mantinha o valor da contagem para ser um *array* de *strings*. Esta variável, que agora tem o nome `operations_history`, irá armazenar o nome das operações que são executadas, e por isso, no *advice* `track_operations` foi alterada a instrução responsável pelo incremento da antiga variável de contagem, para passar a chamar uma função definida na transformação que será responsável por acrescentar o nome da operação, passado como argumento, no *array* `operations_history`.

A função que acrescenta o nome da operação executada à variável com o histórico chama-se `push_operation`. Basicamente a função obtém a posição onde será adicionado o nome da operação, atribuindo-a à variável local `index`, para depois ser utilizada na atribuição do valor do argumento `` `name` `` no respetivo índice do *array* `operations_history`. A atribuição da posição à variável `index` foi feita com recurso às *runtime references*, tanto para obter a referência da variável no primeiro argumento da instrução `local.set`, como para aceder ao tamanho atual do *array* `operations_history` na *runtime expression* definida no segundo argumento da mesma instrução. Contudo, uma vez que este índice não foi invocado num *runtime reference* no interior de um *runtime expression*, e é do tipo primitivo, isto é, é de um tipo já existente na especificação do WASM, faz com que ainda não se encontre registado no módulo do JS (limitação descrita na Secção 3.2). Desta forma, foi necessário registar o índice através da segunda instrução da função. Por último, o nome passado nos argumentos (`/#name/`) é atribuído à variável com o histórico no respetivo índice (`#operations_history[index]`) que foi preenchido anteriormente.

A função `initial_operations_count` foi alterada para passar a ser uma função exportada (com o nome `set_initial_operations_count`) que preenche a variável do histórico com um número de posições preenchidas a `null` (como é do tipo *string* ficará `''`). Este número é passado como argumento à função. As transformações *runtime* implementadas na função são análogas às transformações da função `push_operation`.

A função `get_operations_count` devolve o tamanho do *array* com o histórico e a função `get_operations_history` devolve mesmo o valor do *array*. A primeira utiliza um *runtime expression* para obter o tamanho do *array* (`/#operations_history.length/`), a segunda apenas devolve o *array* através da *runtime reference* (`#operations_history`).

Nesta secção são demonstradas mais funcionalidades associadas ao conceito *runtime*, que incluem, o tipo adicional *array*, o uso avançado de *runtime expressions*, e a introdução das *runtime references*.

4.2.12. *Caching* das Operações

Nesta secção foi implementada a transformação ilustrada na Figura 53, que consiste na implementação de uma *cache* para cada operação existente no programa.

```
pointcuts:
  ...
  internal_operations: () => func(* *(..), internal) && template(operation)
...
aspects:
  context:
    functions:
      log_message:
        args:
          - name: message
            type: string
        imported:
          module: env
          field: log_message
      ...
    variables:
      ...
      operations_cache: "map[string]map[i32]map[i32]i32 = []"
  advices:
    ...
    track_operations:
      pointcut: (i32.local[res] res) => internal_operations()
      all: true
      smart: true
      order: 4
      advice: >
        (call %push_operation% /%'"';operation:select(name);'"'%/)
        (target (local.tee %res% %this%))
        (call %log_operation% /%'"';operation:select(name);'"'%/ (local.get %re
s%))
    add_operations_cache:
      pointcut: () => internal_operations()
      all: true
      smart: true
      order: 3
      variables:
        res: i32 = 0
        lv: i32 = 0
        rv: i32 = 0
      advice: >
        (local.set %lv% %operation:select(first)%)
        (local.set %rv% %operation:select(second)%)
        (block $B0
          (br_if $B0
            (i32.ne
              /!!#operations_cache[%'"';operation:select(name);'"'%] && !!#oper
ations_cache[%'"';operation:select(name);'"'%][#lv] && !!#operations_cache[%'"'
;operation:select(name);'"'%][#lv][#rv]/
              (i32.const 0)
            )
          )
          (local.set %res% %this%)
```

```
          (global.set #operations_cache[%'"';operation:select(name);'"'%][lv][r
v] /#res/)
          (call %log_message% /%'"adding result to cache"'%/)
        )
        (target (local.tee %res% /#operations_cache[%'"';operation:select(name)
;'"'%][#lv][#rv]/))
```

Figura 53 - Código com a transformação onde é implementada uma *cache* nas operações do exemplo da especificação da linguagem

Para esta transformação foi criado o *advice* `add_operations_cache`, cujo *pointcut* é o mesmo que o *pointcut* do *advice* `track_operations` uma vez que ambas têm como objetivo obter as instruções com operações. Desta forma, a expressão foi encapsulada num *pointcut* global com o nome `internal_operations`, que foi invocada nestes *advices*.

A implementação da *cache* baseia-se na criação de uma variável global do tipo mapa (`map[string]map[i32]map[i32]i32`) que armazena o resultado da operação para uma determinada operação e os seus parâmetros. Depois no *advice* é feita a verificação se existe algum valor na *cache*, e caso não exista é feito o cálculo e o resultado é armazenado na *cache*. Uma nota a ter em conta é que as *runtime expressions* não podem ser usadas diretamente na instrução `target`, e por isso, foi adicionada na instrução `local.tee`.

O *pointcut* `track_operations` foi alterado para executar após o *advice* `add_operations_cache`, recebendo nos parâmetros do *pointcut* a variável local `res` criada no *advice* `` `add_operations_cache` ``.

Foi também criada a função importada `log_message` (no campo `log_message` do módulo `env`) que permite o registo de uma determinada mensagem do tipo *string*. Este método é invocado sempre que um dado resultado é inserido na *cache*.

Com a implementação desta transformação foi possível demonstrar a utilização avançada dos conceitos *runtime*, incluindo a utilização do tipo adicional mapa. Além disso, também foi demonstrada a utilização de variáveis locais nos parâmetros passados ao contexto do *pointcut*.

4.2.13. Execução do Exemplo

Para executar o código WASM com a transformação final foi implementado o código JS da Figura 54, estando o resultado da consola ilustrado na Figura 55.

```
const wmr = await import("./output.js");
var instance = (await wmr.loadWasm("./output.wasm", {
env: {
    bef_call: (from, to) => console.log("before", from, to),
    aft_call: (from, to) => console.log("after", from, to),
    log_operation: (op, res) => console.log("operation", op, res, instance.expo
rts.get_operations_count()),
    log_message: (message) => console.info(message)
}
})).instance;
try {
    console.log('running...', instance.exports);
```

```
    instance.exports.set_initial_operations_count(10);
    console.log("initial operations", instance.exports.get_operations_history()
);

    console.log("===");
    console.log("add", instance.exports.math(2, 1, 2));
    console.log("add", instance.exports.math(2, 1, 2));
    console.log("add", instance.exports.math(2, 2, 2));

    console.log("===");
    console.log("sub", instance.exports.math(0, 1, 2));
    console.log("sub", instance.exports.math(0, 1, 2));
    console.log("sub", instance.exports.math(0, 2, 2));

    console.log("===");
    console.log("mul", instance.exports.math(1, 1, 2));
    console.log("mul", instance.exports.math(1, 1, 2));
    console.log("mul", instance.exports.math(1, 2, 2));

    console.log("===");
    console.log("final operations", instance.exports.get_operations_history());
} catch (e) {
    console.error(e);
}
```

Figura 54 - Código JS que executa as transformações realizadas no código do exemplo da especificação da linguagem

```
running... {math: ƒ, set_initial_operations_count: ƒ, get_operations_count: ƒ,
get_operations_history: ƒ}
initial operations (10) ['', '', '', '', '', '', '', '', '', '']
===
before math f1
adding result to cache
operation add 3 11
after math f1
add 3
before math f1
operation add 3 12
after math f1
add 3
before math f1
adding result to cache
operation add 4 13
after math f1
add 4
===
before math f2
adding result to cache
operation sub 1 14
after math f2
sub 1
before math f2
operation sub 1 15
after math f2
sub 1
before math f2
adding result to cache
```

```
operation sub 0 16
after math f2
sub 0
===
before math wmr_f7
adding result to cache
operation mul 2 17
after math wmr_f7
mul 2
before math wmr_f7
operation mul 2 18
after math wmr_f7
mul 2
before math wmr_f7
adding result to cache
operation mul 4 19
after math wmr_f7
mul 4
===
final operations (19) ['', '', '', '', '', '', '', '', '', '', 'add', 'add', 'a
dd', 'sub', 'sub', 'sub', 'mul', 'mul', 'mul']
```

Figura 55 - Resultado na consola da execução do código da especificação da linguagem transformado

No Anexo B encontra-se descrita a especificação completa desta linguagem. Esta especificação aborda a linguagem de uma forma muito mais detalhada, apresentando cada aspeto implementado na mesma.

4.3. Execução da Ferramenta

A execução da ferramenta WasmManipulator é feita através do executável `wmr`. Esta suporta um conjunto de opções que permitem providenciar ao utilizador a capacidade de configurar a execução da mesma. A configuração da ferramenta pode ser feita através de variáveis de ambiente ou da passagem de parâmetros ao iniciar a sua execução. Na Figura 56 encontra-se um exemplo de um comando usado para executar a ferramenta, onde o nome do módulo WASM de entrada é definido numa variável ambiente (`WMR_IN_MODULE="module.wasm"`), e os advices incluídos na execução são definidos no parâmetro de execução (`--include=advice_1,advice_2`).

```
WMR_IN_MODULE="module.wasm" ./wmr --include=advice_1,advice_2
```

Figura 56 - Exemplo do comando para executar a ferramenta WasmManipulator

Na Tabela 5 encontra-se um resumo das configurações existentes na ferramenta. Cada configuração tem um tipo específico e um valor predefinido. A descrição detalhada de cada uma destas opções de configuração encontra-se no Anexo A.

Tabela 5 - Configurações da ferramenta WasmManipulator

	Variável Ambiente	*Parâmetro*	*Tipo*	*Default*
Ficheiro de entrada do módulo	WMR_IN_MODULE	in_module	*string*	input.wasm
Ficheiro de entrada da transformação	WMR_IN_TRANSFORM	in_transform	*string*	input.yml
Ficheiro de saída do módulo transformado	WMR_OUT_MODULE	out_module	*string*	output.wasm
Ficheiro de saída do JS auxiliar	WMR_OUT_JS	out_js	*string*	output.js
Ficheiro de saída do módulo original	WMR_OUT_MODULE_ORIG	out_module_orig	*string*	*null*
Diretoria com as dependências	WMR_DEPENDENCIES_DIR	dependencies_dir	*string*	./dependencies/
Diretoria com dados para execução	WMR_DATA_DIR	data_dir	*string*	./
Ficheiro de logs	WMR_LOG_FILE	log_file	*string*	*null*
Incluir advices	WMR_INCLUDE	include	*string[]*	Todos
Excluir advices	WMR_EXCLUDE	exclude	*string[]*	Nenhum
Gerar sempre o ficheiro JS	WMR_PRINT_JS	print_js	*boolean*	*false*
Gerar sempre o módulo transformado	WMR_ALLOW_EMPTY	allow_empty	*boolean*	*false*
Gerar todo os logs	WMR_VERBOSE	verbose	*boolean*	*false*
Não ordenar advices	WMR_IGNORE_ORDER	ignore_order	*boolean*	*false*

4.4. Integração com o JS

Ao serem utilizadas as funcionalidades *runtime*, a ferramenta para além de gerar o módulo WASM, gera também um ficheiro JS auxiliar. Este ficheiro JS é composto por um módulo JS com o código necessário para a execução das novas funcionalidades. Desta forma, o utilizador precisa de importar este módulo e não o módulo WASM. Para isso, o módulo expõe um método que recebe como argumentos o URI para o ficheiro WASM, e o objeto a importar nesse módulo (opcional). Depois de chamada, a função devolve um objeto com a instância do módulo e a instância com os elementos exportados, à semelhança do método `WebAssembly.instantiate / instantiateStreaming`. Na Figura 57 encontra-se um exemplo com a importação do módulo JS, retirado do exemplo presentes na Secção 6.2.6.1.

```
const wmr = await import('./output.js');
const { instance, model } = (await wmr.loadWasm("./output.wasm", importObject))
;
const wasm = instance.exports;
```

Figura 57 - Código JS com integração do módulo JS produzido pela ferramenta com transformações *runtime*

4.5. Conclusão

Neste capítulo começou-se por analisar os requisitos estabelecidos para a ferramenta WasmManipulator. Depois foi descrito um breve tutorial com exemplos da utilização da ferramenta, onde não só se introduziu a linguagem definida na ferramenta como também se demonstrou a aplicação dos requisitos anteriormente estabelecidos. Seguiu-se uma breve secção sobre a execução da ferramenta, onde foi demonstrada a forma como esta deveria ser iniciada e configurada. Para terminar, existe uma breve secção que trata a integração do módulo JS gerado no código JS cliente.

CAPÍTULO 5.
ARQUITETURA E FLUXO DE DESENVOLVIMENTO

Neste capítulo será abordada a arquitetura da implementação da ferramenta, assim como será detalhado os passos que englobam cada fase existente no fluxo de execução. Este fluxo pode ser dividido nas seguintes fases:

1. Configuração da ferramenta.
2. Leitura dos ficheiros de entrada.
3. Transformação do módulo.
4. Geração de resultados.

Na Figura 58 está ilustrado um esquema com o fluxo associado a estes passos. Nas próximas subsecções estes vão ser detalhados individualmente, descrevendo a implementação para cada um deles, a sua arquitetura e as ferramentas que foram utilizadas para os implementar.

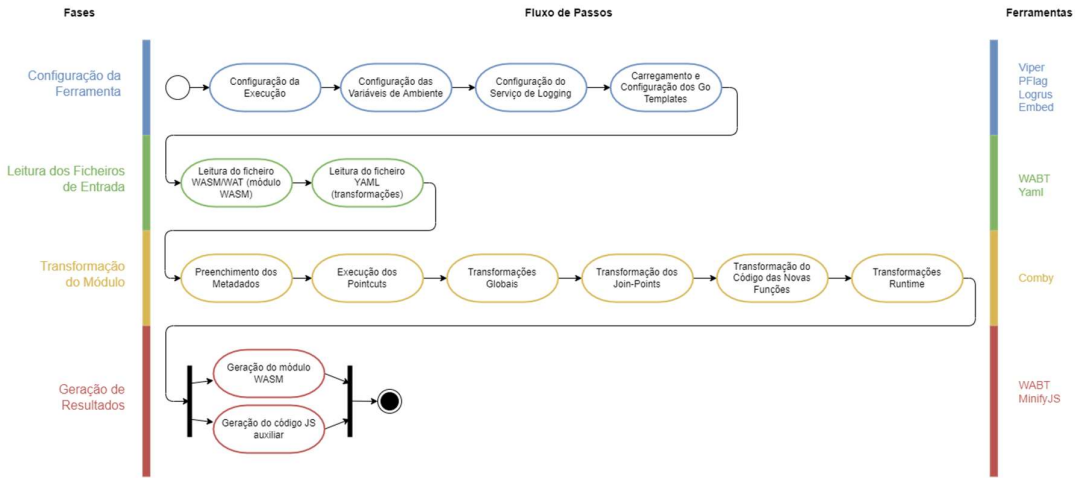


Figura 58 - Fluxo interno com os passos da execução da ferramenta

Este capítulo inicia com a Secção 5.1, onde será apresentada a estrutura de *packages* utilizada no desenvolvimento da ferramenta. Isto porque a ferramenta foi desenvolvida com recurso à linguagem de programação Go, que por definição, estrutura as aplicações através de *packages*.

Depois inicia a descrição do fluxo de execução, começando pela primeira fase, descrita na Secção 5.2, que se baseia na configuração da execução da ferramenta. Esta fase é dividida em quatro subfases, a configuração da execução, a configuração das variáveis de ambiente, a configuração do serviço de *logging*, e por fim, o carregamento e configuração dos *Go Templates*.

Na Secção 5.3 é abordada a segunda fase do fluxo de execução que engloba a leitura dos ficheiros de entrada. Estes ficheiros correspondem ao módulo WASM que se pretende transformar e ao ficheiro YAML com as transformações a realizar.

A Secção 5.4 descreve a forma como as transformações são aplicadas ao módulo, englobando uma série de passos internos que inclui a análise, obtenção de *join-points*, e aplicação das transformações estáticas e *runtime*.

Por fim, a Secção 5.5 consiste numa breve secção que apresenta a forma como os resultados gerados pela ferramenta são processados e armazenados na máquina do utilizador.

5.1. Arquitetura de *packages*

A ferramenta foi desenvolvida com a linguagem de programação Go e encontra-se organizada em diferentes componentes reutilizáveis do código, conhecidos por *packages*, que interagem entre si. Na Figura 59 é possível visualizar o esquema referente à estrutura de *packages* utilizada no desenvolvimento, onde do lado esquerdo temos os *internal packages* e do lado direito os *packages* de carácter geral implementados na ferramenta. As setas representam as interações existentes entre os mesmos.

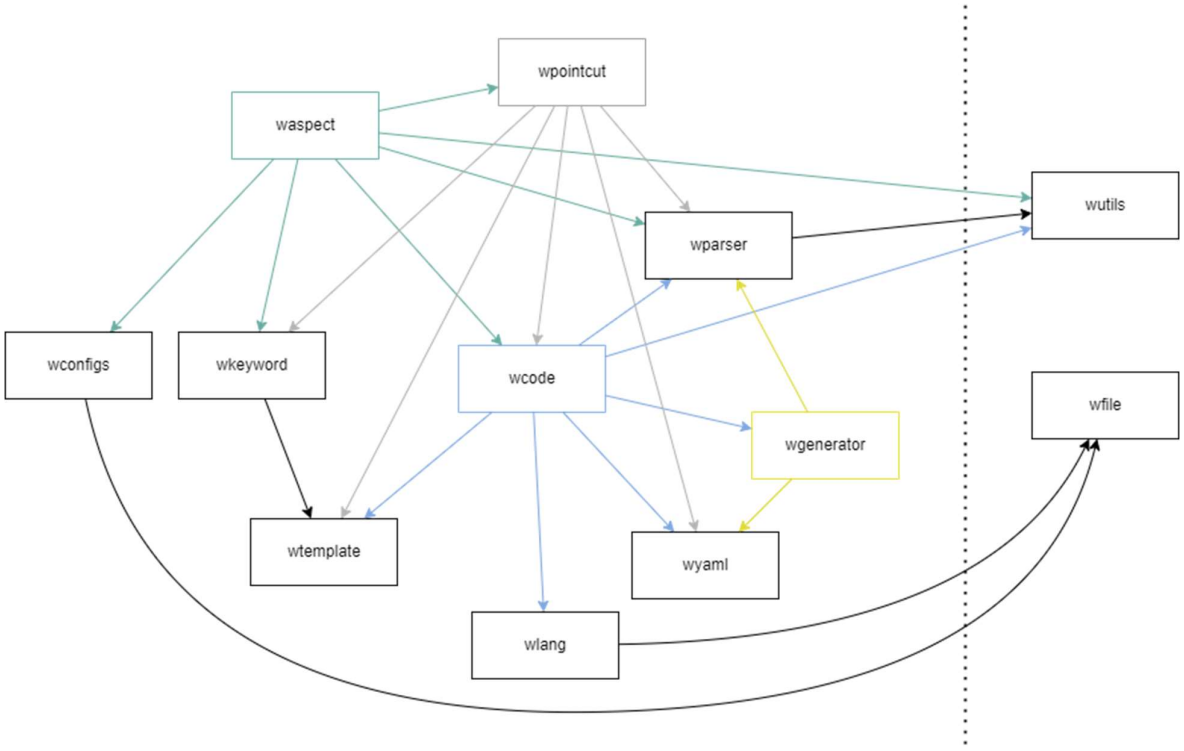


Figura 59 - Esquema arquitetural dos packages na ferramenta

Apesar de possuírem grande parte da lógica de negócio da ferramenta, nenhum deles possui um ponto de entrada que permita que a ferramenta seja devidamente executada. Para isso é necessário a criação do *package* `main`. Este é responsável por obter os dados necessários para a execução, inicializar o processo de transformação sobre esses dados, e gerar os resultados dessa transformação. O código presente neste *package* contém as configurações de execução (Secção 5.2.1) e do serviço de *logging* (Secção 5.2.3). Para além disso, é responsável pelo fluxo de leitura dos ficheiros de entrada (Secção 5.3) e geração dos resultados (Secção 5.5).

O *package* `waspect` é responsável pelo fluxo de transformação (Secção 5.4), isto é, controla todo o processo de uma forma muito superficial, e como consequência deve fazer a gestão de todos os dados necessários para cada passo do fluxo. Este fluxo engloba o preenchimento dos metadados associados ao módulo (Secção 5.4.1), a obtenção dos *join-points* através da execução dos *pointcuts* (Secção 5.4.2), e por fim, a execução das várias transformações ao módulo, que incluem, transformações ao contexto global (Secção 5.4.3), aos *join-points* (Secção 5.4.4), ao código das funções do contexto global (Secção 5.4.5) e às expressões e tipos *runtime* (Secção 5.4.6).

O *package* `wcode` é responsável pela análise de código, execução das *runtime expressions*, recolha de dados de contexto no código, e pesquisa de código que ficará associado aos *join-points*. Possui também a implementação das operações de *parse* (Secção 5.4.1.1) e análise do módulo de entrada (Secção 5.4.1.2), aplicação das transformações (Secções 5.4.3, 5.4.4, e 5.4.6) e gestão dos tipos de dados (Secção 5.4.6.3).

O *package* `wconfigs` faz a gestão de toda a parte de configurações do utilizador na ferramenta. Esta gestão engloba a interpretação e validação das configurações introduzidas pelo utilizador (Secção 5.2.1), quer através de variáveis de ambiente, ou através de argumentos passados na execução. Este disponibiliza um objeto *Singleton* com todas estas configurações, estando disponível para consulta em todos os outros *packages*.

O *package* `wgenerator` tem como objetivo realizar a geração de código WAT. Para isso, recorre aos *Go Templates*, que permitem que o código seja gerado segundo as configurações impostas. Este é responsável por carregar e configurar os *Go Templates* (Secção 5.2.4), e gerar os vários pedaços de código associados a esses *templates*, que poderão ser utilizados durante os diversos tipos de transformação que ocorrem na ferramenta.

O *package* `wkeyword` tem como principal objetivo definir o mecanismo utilizado para o acesso às variáveis dentro das *static expressions*. Para além disto, define dois tipos de "mapas" que implementam este mecanismo, e que permitem a o acesso variáveis do tipo *object* e do tipo *template_search* (detalhados acima na Secção s3.1).

O *package* `wlang` inclui toda a definição das instruções existentes no WASM, onde para cada uma das instruções é especificado os seus tipos e número de parâmetros/resultados. Além disso, disponibiliza um conjunto de funções utilitárias que permitem o acesso e interação com a definição destas instruções.

O *package* `wparser` é composto pelos *parsers* responsáveis por interpretar elementos textuais que não são do tipo WAT. Os elementos textuais que necessitam de ser interpretados para além do código WAT são as *static expressions*, as expressões presentes no *pointcut* (Secção 5.4.2.2), os *templates* e a declaração/inicialização das variáveis.

O *package* `wpointcut` é responsável por executar a árvore sintática abstrata proveniente da expressão do *pointcut*. Esta execução tem como objetivo obter os *join-points* para mais tarde aplicar as transformações. Desta forma, este *package* é utilizado durante o processo de

execução dos *pointcuts* (Secção 5.4.2), onde para além de obter os *join-points* (Secção 5.4.2.3), também faz toda a gestão dos dados de contexto provenientes da expressão.

O *package* `wtemplate` controla todas as ações internas associadas à funcionalidade dos *templates* na ferramenta. Esta é responsável pela sua execução, validação, combinação de *templates*, e armazenamento de dados. Estas operações são efetuadas durante o processo de obtenção dos *join-points*, e posteriormente para acesso nas *static expressions*.

O *package* `wyaml` é responsável pelo modelo que armazena o código de transformação, obtido através da leitura do respetivo ficheiro de entrada. Internamente recorre a ferramentas próprias da linguagem Go que permitem o *parse* desse ficheiro para um modelo interno (Secção 5.3).

Os *packages* `wfile` e `wutils` são de caráter geral, sendo que o `wfile` contém várias funções utilitárias que permitem abstrair a interação e manipulação com o sistema de ficheiros, enquanto que o `wutils` é composto por várias funções também utilitárias, no entanto de carácter geral, tais como, manipulação de *strings*, comparação de valores, estruturas de dados abstratos, etc.

5.2. Configuração da Ferramenta

Esta fase corresponde essencialmente à configuração geral da ferramenta, que inclui a definição das variáveis de ambiente necessárias à execução da ferramenta, as configurações provenientes do utilizador, definidas através de variáveis de ambiente ou através de argumentos passados na execução, a configuração do serviço de *logging*, e por fim, o carregamento e compilação dos *Go Templates*.

Cada subsecção vai corresponder a um dos sub-passos da configuração. As subsecções vão estar na ordem correspondente ao fluxo da execução. Desta forma, a Secção 5.2.1 trata das configurações do utilizador, onde são descritas as bibliotecas utilizadas no tratamento destas configurações, e a forma como são armazenadas e acessadas durante a execução. Existem diversas configurações disponibilizadas ao utilizador, estando estas especificadas na Secção 4.2. Depois existe a Secção 5.2.2 para a configuração das variáveis de ambiente que são utilizadas pela ferramenta. A Secção 5.2.3 que se segue apresenta um breve resumo da forma como é configurado o serviço de *logging* na ferramenta. Por fim, a Secção 5.2.4 contém toda a informação sobre a forma como são integrados os *Go Templates* na ferramenta.

5.2.1. Configuração da Execução

Para a configuração da execução da ferramenta realizada pelo do utilizador, foi disponibilizado um conjunto de configurações que podem ser definidas através de variáveis de ambiente ou dos argumentos passados aquando a execução. A integração de ambas as formas foi realizada com recurso às bibliotecas Viper (Contributors V. , 2021) e PFlag (Contributors P. , 2021).

Essencialmente, a biblioteca Viper foi a biblioteca responsável por quase toda a parte de gestão destas configurações na ferramenta. A única exceção foi a obtenção das configurações via argumentos de execução, que foi responsabilidade da biblioteca PFlag,

onde mesmo assim, passou todos os valores obtidos nos argumentos para a biblioteca Viper. Os valores assumidos para cada uma destas configurações passam por três etapas distintas. Na primeira etapa, a ferramenta assume que as configurações possuem um valor pré-definido, e configurado internamente no código-fonte. Estas configurações podem ser sobrepostas pelas configurações definidas nas variáveis de ambiente. Por fim, pode existir uma nova sobreposição, mas desta vez com as configurações inseridas via argumentos de execução.

Estas configurações são definidas diretamente na biblioteca Viper que por sua vez cria um *Singleton* interno (Contributors V. , 2021) que permite a sua gestão. Com os valores finais definidos no Viper, é feito o *decode* (função *unmarshal*, frequentemente utilizado na linguagem Go) das configurações para uma estrutura interna no código da ferramenta, contendo o valor para cada uma dessas configurações.

Todo este processo, incluindo a gestão e armazenamento do objeto de configurações, encontra-se isolado no *package* `wconfigs`. Desta forma, as configurações podem ser acessadas em qualquer parte da ferramenta através deste.

5.2.2. Configuração das Variáveis de Ambiente

Atualmente, a única variável de ambiente alterada pela ferramenta consiste no *PATH*, onde são adicionados os caminhos para o executável de cada uma das dependências externas ao ambiente da linguagem, o WABT, o MinifyJS e o Comby.

Caso a variável de ambiente *WMR_DEPENDENCIES_DIR* (configuração especificada na Secção 4.2) seja definida pelo utilizador, o caminho para os executáveis tem como base essa diretoria. Caso contrário, é assumido o caminho onde foi executada a ferramenta. A diretoria "dependencies/" (presente na raiz do projeto) deverá existir no caminho configurado, onde deve possuir a estrutura de ficheiros ilustrada na Figura 60, incluindo também os executáveis. No caso do utilizador possuir todas as dependências já instaladas e configuradas na variável de ambiente *PATH*, estas configurações podem ser ignoradas. Contudo, esta opção pode criar problemas na execução da ferramenta, uma vez que as versões instaladas podem não ser compatíveis com as mesmas.

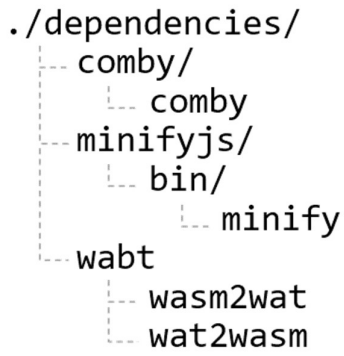


Figura 60 - Estrutura da diretoria com as dependências da ferramenta ("dependencies/")

5.2.3. Configuração do Serviço de *Logging*

O serviço de *Logging* utilizado pela ferramenta está contido na biblioteca Logrus (Contributors L. , 2021), que tal como a biblioteca Viper, implementa uma instância interna do tipo *Singleton*. A diferença deste serviço para o de configurações, é que ao invés de ser armazenado internamente, é sempre acedido, pela ferramenta, de forma direta à instância na biblioteca.

A configuração deste serviço engloba a definição do nível de *logging*, o destino para o registo dos *logs*, e o formato de cada registo. Tanto a definição do nível como o destino dos *logs* estão definidos nas configurações da execução (Secção 5.2.1).

Por predefinição, o nível de *logging* é o `Info`, o que significa que apenas são registados os *logs* de informação, aviso ou erro. Os *logs* de rastreamento e *debugging* apenas são registados se a configuração *verbose* (Secção 4.2) estiver ativa.

O registo de *logs* por predefinição é feito para a linha de comandos onde foi executada a ferramenta, no entanto, também é possível configurar o serviço para fazer os registos num dado ficheiro. Neste último caso a ferramenta é responsável por criar o ficheiro no caminho indicado na respetiva configuração, sobrepondo qualquer outro ficheiro com o mesmo caminho (caso exista), e por definir este ficheiro como fonte principal de saída no serviço.

5.2.4. Carregamento e Configuração dos *Go Templates*

A ferramenta utiliza *Go Templates* na geração de código WAT. Para isso foi criado um conjunto de *templates* que após compilação podem ser importados com as devidas configurações e utilizados na ferramenta. Os *templates* criados podem ser divididos em duas categorias: elementos e instruções WAT e instruções WAT para transformações *runtime*.

Os *templates* que permitem a geração de código para elementos e instruções WAT são principalmente utilizados para adicionar os novos elementos definidos no ficheiro de transformação ao módulo. Tal grupo engloba os elementos para funções (incluindo função inicial, importadas e exportadas), para variáveis locais e globais, e instruções para alterar o valor dessas variáveis. Relativamente aos *templates* que geram o código WAT para transformações *runtime*, estes são responsáveis por gerar o código associados aos diferentes comportamentos da funcionalidade. Com isto, existem diversos *templates* criados na ferramenta que são gerados de acordo com o contexto onde as *runtime expressions* são executadas ou o local onde são aplicados os tipos adicionais ao WAT. O contexto das *runtime expressions* é definido de acordo com o tipo do resultado da expressão, ou seja, se o tipo esperado existe no WAT ou é um tipo adicional, de acordo com o tipo de instrução onde foram invocadas, e de acordo com a posição onde as instruções serão adicionadas, isto é, se são para adicionar antes, invés, ou depois da respetiva instrução. Quanto às modificações dos tipos adicionais, estes dependem do local onde se encontram, por exemplo, se são parte de uma variável global, parâmetro de uma função, tipo de retorno, etc.

Como a complexidade dos *templates* desta segunda categoria é bastante superior à complexidade dos da primeira, foi necessário isolar cada *template* num ficheiro individual.

Para que isso fosse possível, foi necessário a utilização da biblioteca *Embed*, que permitiu carregar o conteúdo dos ficheiros em tempo de compilação, e preencher as respetivas variáveis presentes no código-fonte com o mesmo.

5.3. Leitura dos Ficheiros de Entrada

O objetivo da ferramenta é a realização de transformações a um módulo WASM, e por isso, para executar são esperados dois ficheiros de entrada: ficheiro com o módulo WASM (".wasm" ou ".wat") e o ficheiro de transformação (".yml").

A leitura do ficheiro com o módulo WASM engloba a utilização de outra ferramenta, o WABT. Nesta fase, esta ferramenta não só é necessária para que o conteúdo lido pela aplicação esteja sempre num formato WAT, mas também para que a sua estrutura seja consistente. Esta consistência é necessária para que o *parser* implementado internamente esteja familiarizado com a sintaxe do código WAT referente ao módulo.

Caso o utilizador introduza um ficheiro do tipo WAT, para que o conteúdo seja válido, será necessário primeiro transformar o conteúdo para WASM e só depois converter novamente para WAT. Para isso, será necessário primeiro executar o comando `wat2wasm` sobre o ficheiro de entrada, e depois o comando `wasm2wat` sobre o ficheiro que resultou do primeiro comando (ficheiro WASM). Se for introduzido um ficheiro do tipo WASM, apenas é necessário executar o segundo comando sobre o ficheiro de entrada. Depois disto, a ferramenta irá ler o ficheiro resultante (com código WAT válido), e posteriormente apagá-lo do sistema (no primeiro caso).

Relativamente ao ficheiro de transformações, este deve ter o formato YAML especificado na Secção 1 do Anexo B, e engloba apenas a biblioteca *yaml* para efetuar o *decode* do conteúdo do ficheiro para uma estrutura de dados interna. Esta operação é inteiramente realizada no *package* `wyaml`, no qual também possui toda a definição do modelo de dados para o conteúdo.

5.4. Transformação do Módulo

A transformação do módulo ocorre segundo uma definição expressa no ficheiro de transformações. Esta transformação é dividida em vários passos, executados segundo uma determinada ordem, sendo que o resultado final será o código WAT transformado e, se for o caso, o código JS auxiliar.

Numa fase inicial do desenvolvimento, parte deste processo de transformação era realizado com recurso a uma ferramenta externa conhecida por WASM-JSON-Toolkit (Contributors W.-J.-T. , 2021), que tem como objetivo converter WASM em JSON e vice-versa. Desta forma, o módulo WASM a transformar era convertido para JSON através desta ferramenta, e posteriormente convertida num modelo interno pronto a ser manipulado pela ferramenta WasmManipulator. Contudo, para além desta abordagem ser muito pouco eficiente, uma vez que para cada transformação seria necessário efetuar o processo de escrita, leitura e execução da ferramenta WASM-JSON-Toolkit, provocando assim sérios problemas de desempenho, com a inserção das funcionalidades de *static* e *runtime expressions*, esta tornou-se

inexequível uma vez que os elementos presentes nas expressões eram completamente desconhecidos para a ferramenta. Desta forma, foi necessário alterar todas as funcionalidades de análise do módulo e inserção de elementos para uma implementação que suportasse este tipo de expressões. Como consequência da nova implementação, não só o WasmManipulator passou a suportar todo o tipo de modificações previstas, como também aumentou drasticamente o desempenho da execução e simplificou bastante o processo de desenvolvimento.

As subsecções que se seguem vão abordar com maior detalhe cada um destes passos. A Secção 5.4.1 está relacionada com o preenchimento dos metadados, feito no início da transformação, e que servem de apoio para cada um dos seguintes passos. Depois, na Secção 5.4.2 será detalhado o processo de execução dos *pointcuts* para cada *advice* definido pelo utilizador. Esta execução tem como objetivo a obtenção dos *join-points* que serão utilizados para realizar a transformação no *advice*. Segue-se a aplicação das transformações relativas ao contexto global do módulo (Secção 5.4.3). Estas transformações englobam a criação de novas funções, tipos e variáveis globais. Na Secção 5.4.4 vai ser detalhado o processo de aplicação das transformações aos *join-points* obtidos em passos anteriores, através da execução de *static expressions*. Segue-se a Secção 5.4.5 que está relacionado com a transformação do código das novas funções adicionadas pelo utilizador. Este visa não só abordar o porquê de existir uma separação do passo em que são adicionadas, mas também da forma como são efetuadas as devidas transformações. E por fim, a Secção 5.4.6 aborda como se procedem as transformações *runtime*, incluindo a execução das *runtime expressions*.

5.4.1. Preenchimento dos Metadados

O preenchimento dos metadados que são necessários ao processo de transformação inicia com a conversão do código WAT para uma estrutura semelhante a uma AST. Esta conversão é feita através de um *parser* que tem como objetivo analisar o código, e à medida que essa análise é feita, traduzir as instruções para o respetivo *token*. Depois disso, é aplicado o padrão de *design Visitor* para obter toda a informação necessária sobre o módulo. Com esta informação, é inicializada a estrutura referente ao metadados, que durante a fase de transformação servirá como referência para diversas operações e poderá ser alterada de acordo com o tipo de transformação aplicada.

5.4.1.1. *Parse* do Código WAT

Como foi mencionado acima, o *parse* do código WAT é feito por um *parser* interno. Assim, foi implementado um *Scannerless Parser* (Tomassetti, 2017) pelo facto de ter sido considerado que a análise do código WAT, no âmbito da ferramenta, não possuía complexidade suficiente para justificar a implementação de um *lexer* separado do *parser*. Desta forma, o *parser* é responsável por analisar o conteúdo WAT, transformá-lo numa sequência de *tokens*, e gerar a estrutura de dados necessária a partir dessa sequência.

Na Figura 61 está ilustrado um exemplo com a aplicação do *parser* sobre um bloco de código retirado do exemplo "*Cache* Múltipla" da Secção 6.2.6.2. Neste exemplo estão a ser produzidos os seguintes tipos de *token:*

- ***element*** – Elemento que permite agrupar um conjunto de *tokens*. Neste exemplo possui quatro *tokens*.
- ***instruction*** – Representa uma instrução WAT. Encontra-se sempre contida entre parênteses .
- ***keyword*** – Consiste numa *static expression*.
- ***text*** – Texto simples.
- **evaluation** – Consiste numa *runtime expression*.
- ***evaluationText*** – Bloco de texto inserido numa *runtime evaluation*.
- ***evaluationIndex*** – Valor do índice que permite o acesso a membros para de uma variável dentro das *runtime references*.
- ***evaluationQuoted*** – Valor textual considerado uma *string* dentro das *runtime evaluations*.
- ***evaluationRef*** – Consiste numa *runtime reference*.

Figura 61 - Resultado do *parser* interno após análise de código de transformação

Cada *token* implementa uma *interface* comum que não só permite a manipulação do mesmo, como também percorrer a respetiva AST onde estão incluídos e alterar o formato WAT desejado para o *token*.

5.4.1.2. Análise do Módulo

Após a conclusão do processo de *parsing*, onde foi obtida a AST correspondente ao módulo inserido na ferramenta, é necessário executar a sua análise. Esta análise visa obter

informações essenciais sobre o módulo para processo de transformação que ocorrerá numa fase posterior da ferramenta.

Esta análise foi realizada com recurso ao padrão de *design Visitor*, que consiste num padrão muito utilizado para este tipo de análise (Bohm, 2014). Para isso, foram definidos quatro *Visitors*. Apesar de cada um possuir a sua própria função, estes implementam todos a mesma *interface*, permitindo assim que a AST seja percorrida pelos diferentes *Visitors*. Desta forma, à medida que a AST é percorrida, cada *Visitor* é acionado de acordo com o tipo de elemento, e de acordo com a sua função, atualizam devidamente uma estrutura específica com a informação do módulo. No final, esta estrutura deverá conter toda a informação necessária para se proceder à transformação do módulo.

Foram criados Visitors para recolher a informação dos tipos, das funções, das variáveis globais e dos elementos que são exportados. A ordem pelo qual foram executados importa, uma vez que uma certa análise pode necessitar da informação proveniente da análise anterior. Desta forma, é importante referir que a ordem apresentada acima corresponde à ordem de execução na ferramenta.

Tendo sido concluída a análise, devem ser conhecidos os dados relativos às funções e às variáveis globais presentes no módulo. A informação relativa às funções inclui se a função é exportada, importada ou nenhuma das duas, o seu tipo (protótipo da função), parâmetros, variáveis locais e instruções do código. Relativamente à informação sobre as variáveis globais, esta inclui o seu nome, tipo, valor inicial, se é mutável ou não, e se é exportada, importada ou nenhuma das duas. Para além disto, existe um conjunto de dicionários definidos na estrutura que servem para relacionar os identificadores declarados no ficheiro de transformação com os índices/nomes dos elementos existentes no módulo. Estes dicionários neste preciso momento encontram-se vazios, contudo serão preenchidos numa fase posterior. A estrutura possui ainda dados relacionados com as transformações *runtime*, no entanto, tal como os dicionários, só serão preenchidos e utilizados mais tarde.

5.4.2. Execução dos *Pointcuts*

A execução dos *pointcuts* tem como principal objetivo obter os *join-points* que depois irão sofrer as modificações definidas no *advice*. Estes *join-points* correspondem a pedaços de código que possuem na sua estrutura algum contexto adicional. A execução dos *pointcuts* é feita de forma individual e separada para cada *advice* definido no ficheiro de transformações. Esta é feita de forma sequencial e segue uma determinada ordem configurada pelo utilizador.

As subsecções que se seguem correspondem às sub-fases que compõem o processo de execução dos *pointcuts*. A Secção 5.4.2.1 é referente à filtragem dos *advices* que serão executados na ferramenta. A próxima (Secção 5.4.2.2) será responsável por apresentar a forma como é feito o *parse* dos *pointcuts*. E por último, existe a Secção 5.4.2.3 que possui os detalhes sobre a obtenção dos *join-points*.

5.4.2.1. Filtragem dos *Advices*

Inicialmente os *advices* a aplicar na ferramenta são filtrados. Esta filtragem engloba não só as configurações introduzidas na ferramenta, mas também um processo de validação da expressão do *pointcut* declarada no *advice*. Caso um *advice* tenha sido excluído pelo utilizador ou possua uma expressão inválida no *pointcut*, este é completamente ignorado pela ferramenta.

5.4.2.2. Parse dos Pointcuts

Após a filtragem dos *advices* é feito o *parse* do *pointcut* presente em cada um. Uma vez que não há necessidade de um controlo apertado sobre o processo de análise e transformação da expressão em *tokens* (*lexing*), ficou decidido que seria desejável utilizar uma biblioteca que facilitasse este processo.

Desta forma, foi utilizada a biblioteca (Contributors P. , 2021) que para além de servir de *lexer*, serve também de *parser*, organizando os *tokens* produzidos num modelo de dados definido na ferramenta. Apesar deste modelo ser de fácil interpretação, refletindo a estrutura do conteúdo quase como uma tradução, não é adequado para a execução das expressões presentes no *pointcut*. Assim, foi criado um *parser* interno simplificado que, utilizando os dados que compõem o modelo, cria uma AST capaz de ser manipulada e executada pela ferramenta.

Como a expressão é considerada como sendo *infix* (operadores encontram-se entre os operandos), foi adaptado um algoritmo conhecido por *Expression Evaluation* (Team G. , Expression Evaluation, 2021), em que com recurso a uma tabela de procedência, foi criada uma AST que permita a execução da esquerda para a direita. Assim, ao executar a AST, os vários *tokens* que a formam são percorridos *inorder* (esquerda, raiz, direita). Após a AST ter sido percorrida, são devolvidos os respetivos *join-points* resultantes da execução (capítulo abaixo).

5.4.2.3. Obtenção dos *Join-Points*

Como foi mencionado na Secção 5.4.2.2, a execução da AST correspondente à expressão de um *pointcut* devolve um conjunto de *join-points*. Para isso, cada *token* presente na AST deve possuir um mecanismo que permita, de acordo com o seu tipo e configuração, a procura dos *join-points* no módulo.

Na Figura 62 encontra-se ilustrado um exemplo da AST que resulta do *parse* da seguinte expressão de *pointcut*: `(func(void sin(..)) || func(void *(..), internal) && template(t, true)) && (returns(void) || call(void *(..)))`. Neste caso o *template* t declara que no código da função devem existir quatro variáveis locais definidas do tipo inteiro *32-bit*.

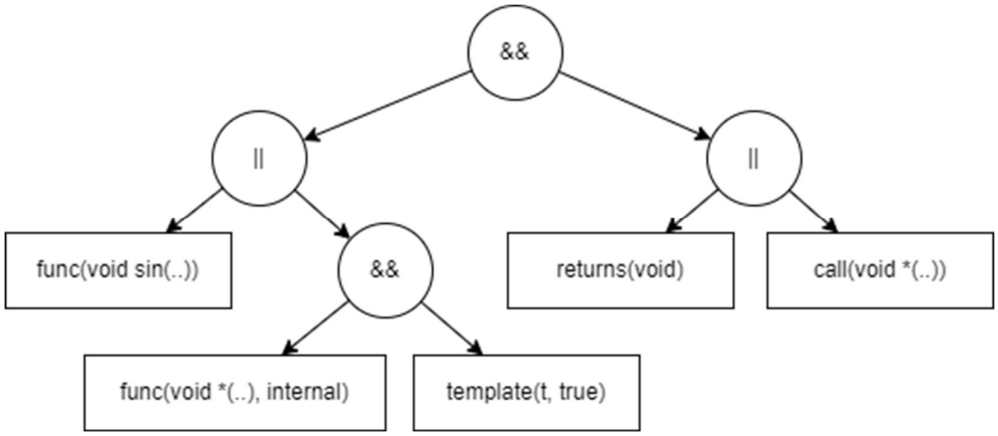
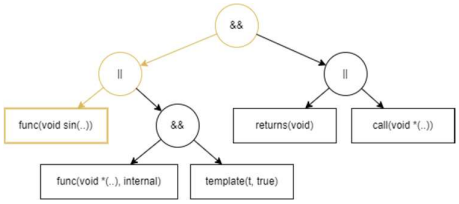
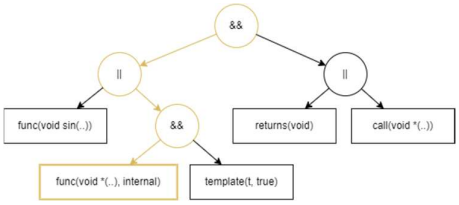
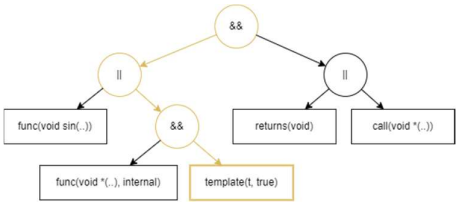
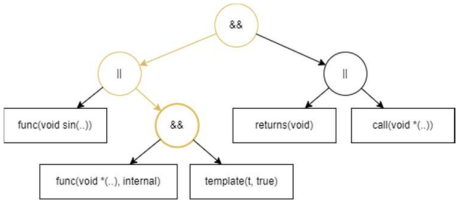


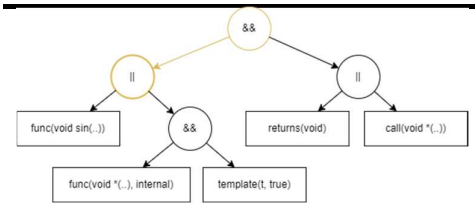
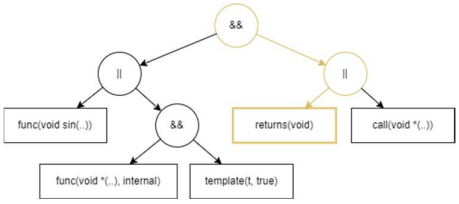
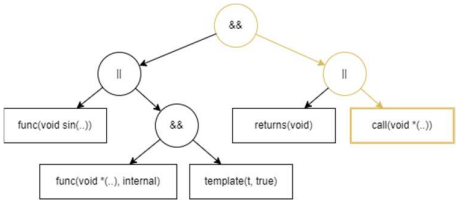
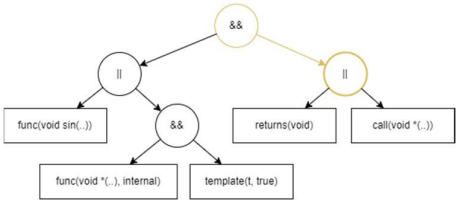
Figura 62 - Árvore sintática resultante do *parse* de uma expressão de *pointcut*

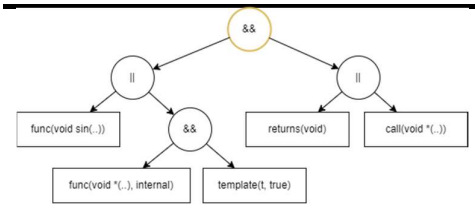
O mecanismo implementado pela ferramenta para efetuar a procura de *join-points* é baseado numa abordagem *top-down*. Isto significa que inicialmente cada função é considerada um *join-point*. Com a execução do *pointcut*, à medida que os *tokens* vão sendo executados, estes *join-points* vão sendo analisados e devidamente filtrados. A Tabela 6 apresenta o fluxo do processo da filtragem dos *join-points*. No final, restam apenas os *join-points* que satisfazem a expressão do *pointcut*.

Tabela 6 - Fluxo de obtenção dos join-points

Expressão	***Execução***
Entrada	**--- 1. Função $f0** (func $f0 (export "sin") (param i32) (local $l0 i32) (local $l1 i32) (block $B0 (...) (return)) (loop $L0 (...))) **--- 2. Função $f1** (func $f1 (param i32 i32) (local $l0 i32) (local $l1 i32) (local $l2 i32) (block $B0 (...) (call $f3)) (return)) **--- 3. Função $f2** (func $f2 (param i32 i32 i32) (local $l0 i32) (local $l1 i32) (local $l2 i32) (local $l3 i32) (block $B0 (...) (call $f0 ...))) **--- 4. Função $f3** (func $f3 (result i32) (local $l0 i32) (local $l1 i32) (local $l2 i32) (local $l3 i32) (...) (call $f2 ...))

Expressão	*Execução*
func(void sin(..)) 	**Entrada:** (func $f0 (export "sin") (param i32) ...) (func $f1 (param i32 i32) ...) (func $f2 (param i32 i32 i32) ...) (func $f3 (result i32) ...) **Saída:** (func $f0 (export "sin") (param i32) ...) | **FuncData** - - -
func(void *(..), internal) 	**Entrada:** (func $f0 (export "sin") (param i32) ...) (func $f1 (param i32 i32) ...) (func $f2 (param i32 i32 i32) ...) (func $f3 (result i32) ...) **Saída:** - (func $f1 (param i32 i32) ...) | **FuncData** (func $f2 (param i32 i32 i32) ...) | **FuncData** -
template(t, true)) 	**Entrada:** (func $f0 (export "sin") (param i32) ...) (func $f1 (param i32 i32) ...) (func $f2 (param i32 i32 i32) ...) (func $f3 (result i32) ...) **Saída:** - - (func $f2 (param i32 i32 i32) ...) | **Template(t)** (func $f3 (result i32) ...) | **Template(t)**
func(void *(..), internal) && template(t, true)) 	**Entrada:** (func $f1 (param i32 i32) ...) | **FuncData** (func $f2 (param i32 i32 i32) ...) | **FuncData** (func $f2 (param i32 i32 i32) ...) | **Template(t)** (func $f3 (result i32) ...) | **Template(t)** **Saída:** - - (func $f2 (param i32 i32 i32) ...) | **FuncData**, **Template(t)** -
func(void sin(..)) || func(void *(..), internal) && template(t, true))	**Entrada:** (func $f0 (export "sin") (param i32) ...) | **FuncData** (func $f2 (param i32 i32 i32) ...) | **FuncData**, **Template(t)** **Saída:** (func $f0 (export "sin") (param i32) ...) | **FuncData** - (func $f2 (param i32 i32 i32) ...) | **FuncData**, **Template(t)** -

Expressão	***Execução***
	
returns(void) 	**Entrada:** (func $f0 (export "sin") (param i32) ...) (func $f1 (param i32 i32) ...) (func $f2 (param i32 i32 i32) ...) (func $f3 (result i32) ...) **Saída:** (return) x2 - dentro de $B0 e no fim (return) - no fim (return) - no fim, uma vez que o *return* é omitido (return) - no fim, uma vez que o *return* é omitido *Como a função não devolve nenhum valor, é inserida automaticamente a instrução return. Caso contrário, seria selecionada a instrução de retorno.*
call(void *(..)) 	**Entrada:** (func $f0 (export "sin") (param i32) ...) (func $f1 (param i32 i32) ...) (func $f2 (param i32 i32 i32) ...) (func $f3 (result i32) ...) **Saída:** - - (call $f0 ...) | **CallData** (call $f1 ...) | **CallData**
returns(void) || call(void *(..)) 	**Entrada:** (return) x2 - $f0, dentro de $B0 e no fim (return) - $f1, no fim (return) - $f2, no fim (return) - $f3, no fim (call $f0 ...) - $f2 | **CallData** (call $f1 ...) - $f3 | **CallData** **Saída:** (return) x2 (return) (call $f0 ...) + (return) | **CallData** (call $f1 ...) + (return) | **CallData**
(func(void sin(..)) || func(void *(..), internal) && template(t, true)) && (returns(void) || call(void *(..)))	**Entrada:** (func $f0 (export "sin") (param i32) ...) | **FuncData** (func $f2 (param i32 i32 i32) ...) \| **FuncData**, **Template(t)** (return) x2 - *$f0* (return) - *$f1* (call $f0 ...) - *$f2* + (return) - *$f2* | **CallData** (call $f1 ...) - *$f3* + (return) - *$f3* | **CallData**

Expressão	***Execução***
	**Saída:** (return) x2 | **FuncData** - (call $f0 ...) + (return) | **FuncData**, **Template(t)**, **CallData** -
Saída	**--- 1. Função $f0 | FuncData** (func $f0 (export "sin") (param i32) (local $l0 i32) (local $l1 i32) (block $B0 (...) (return)) (loop $L0 (...))) **--- 2. Função $f2 | FuncData, Template(t), CallData** (call $f0 ...) -- (return)

Ao longo deste processo, os vários *tokens* referentes às funções do *pointcut* vão acrescentando informações sobre os resultados ao contexto do *join-point* (detalhados na secção da especificação, Secção 2 do Anexo B). Com isto, é possível concluir que os *join-points* são meras estruturas de dados associados a um dados conjunto de instruções, e que contêm dados de contexto relativos aos *pointcuts* que os originaram e ao ambiente a que essas instruções pertencem.

Depois desta breve explicação acerca da execução dos *pointcuts*, é necessário perceber de que forma cada *token* existente na expressão realmente é capaz de obter os respetivos *join-points*.

Cada *token* implementa um método comum que recebe uma estrutura de dados e devolve uma nova estrutura de dados do mesmo tipo, mas modificada. Esta estrutura de dados contém não só o grupo de *join-points* que fora encontrado até àquele ponto, como também os dados necessários para efetuar a filtragem. Adicionalmente, estes dados também são compostos pela estrutura com a informação do módulo previamente carregada (passo descrito na Secção 5.4.1), e a estrutura responsável pela gestão dos *templates*, que é carregada à medida que o processo de execução dos *pointcuts* ocorre.

O *token* de operações é relativo às operações lógicas `AND` e `OR`. Estas operações têm como objetivo a interseção e a união de *join-points*, respetivamente, incluindo não só as instruções do código, como também os dados de contexto preenchidos até ao momento.

Os *tokens* relativos às funções `func`, `call`, `args` e `returns` filtram os *join-points* de forma muito similar. O seu processo engloba novamente a aplicação do padrão de *design Visitor*. onde foi criado um *Visitor* para cada uma das funções, que tem como objetivo detetar instruções do tipo requisitado pelo tipo da função (por exemplo, a função *call* procura instruções que coincidam com chamadas a funções), validar essas instruções com a configuração definida pelo utilizador na respetiva função, e por fim, se for o caso, criar o

join-point associado às instruções, já com o contexto preenchido. Este *Visitor* é sempre aplicado sobre os *join-points* registados anteriormente, causando assim o efeito de filtragem. Todo o processo de gestão de duplicados é feito no final da execução de cada *token*.

Relativamente ao *token* para a função `template`, ao contrário dos anteriores, este necessita de executar uma ferramenta externa própria para a pesquisa por padrão, o *Comby*. O processo de filtragem neste *token* resume-se à execução do respetivo *template* sobre os *join-points* registados anteriormente. Esta execução engloba a construção dos metadados associados ao *template*, tais como, as variáveis declaradas, e as operações adjacentes às variáveis, depois engloba a conversão do *template* introduzido no ficheiro de transformações para um *template* com a sintaxe do *Comby*, e por fim, a ferramenta é executada sobre o código de cada *join-point*. O resultado gerado após cada execução é imprimido num ficheiro que depois será lido, interpretado e convertido para um modelo interno específico. O último passo será validar este resultado, sendo este da responsabilidade das operações que são aplicadas nas *template keywords*. Esta validação consiste em verificar se todas as variáveis estão devidamente declaradas, independentemente do *template* a que pertencem. Após obter os resultados, estes são fundidos com os resultados encontrados até àquele ponto, e convertidos para novos *join-points*. Esta conversão consiste na obtenção das instruções do módulo que coincidam com o resultado textual obtido.

Como a filtragem para o *token* associado ao *template* poderá ser um pouco exaustivo, a implementação foi feita de forma a que o processo de filtragem dos diversos *join-points* seja executada em simultâneo, isto é, cada *join-point* filtrado será executado de forma assíncrona numa *Go Routine*.

5.4.3. Transformações Globais

A ordem pela qual o processo de transformação dos elementos globais é executado poderá variar segundo o estado do campo `All` no *advice*. Caso não esteja ativo, este processo é executado numa fase inicial, uma vez que a obtenção dos *join-points* deve apenas estar associada ao código pré-existente no módulo. Caso contrário, as transformações ocorrem posteriormente, mesmo antes da execução dos *pointcuts*, para que a obtenção dos *join-point* seja também feito sobre o código transformado (já com os elementos adicionados através da ferramenta).

Basicamente esta fase consiste na aplicação de todos os novos elementos, presentes no ficheiro de transformações, ao módulo. Tais transformações englobam a criação de variáveis globais e funções.

A criação de variáveis globais inicia com o *parse* da variável declarada no ficheiro de transformações. Este *parse*, tal como no *parse* dos *pointcuts* (capítulo acima), é realizado com recurso à biblioteca Participle. Já com a declaração/inicialização incluída num modelo interno da ferramenta, é feita a conversão dessa estrutura de dados para código WAT. É importante referir que este tipo de conversão é sempre feito com recurso aos *Go Templates*, que se encontram implementados no *package* `wgenerator`. Depois de gerar o código WAT, para que o elemento seja adicionado ao módulo, é necessário fazer o *parse* desse código para

o mesmo tipo de *tokens* utilizados no modelo interno do módulo. Desta forma, o *parse* do código é realizado da mesma forma que o *parse* realizado inicialmente para o módulo, presente na Secção 5.4.1. Após ter sido feito o *parse*, a informação referente à variável global é inserida no contexto do módulo através de uma análise implementada com recurso ao padrão de *design Visitor* (semelhante ao que foi feito para o módulo).

No que toca à criação de funções, o processo é equivalente à criação de variáveis globais detalhado acima. Este processo poderá também englobar a criação de uma função inicial caso tenha sido definido no ficheiro de transformações código para ser executado inicialmente.

O primeiro passo na criação de uma função consiste na inserção do respetivo elemento *type* caso ainda não exista nenhum para a configuração da função. A sua criação engloba gerar o código WAT associado e fazer o *parse* do mesmo. Os *tokens* gerados pelo *parser* são inseridos no modelo interno do módulo. Este *type* será utilizado na declaração da função.

Segue-se o passo responsável pelo *parse* da função. Cada função declarada no ficheiro de transformações é constituída por vários elementos. Estes são os parâmetros, o tipo do resultado, as variáveis locais, o código do corpo, e a definição de importado ou exportado. Como consequência, é feito o *parse* individual para cada um destes elementos. Depois de serem obtidas todas as estruturas de dados, é realizada a sua conversão em código WAT. Este representa o código WAT da função com todos os elementos inseridos exceto as variáveis locais, que serão adicionadas numa fase posterior, pois podem necessitar de ser inicializadas no corpo da função. Depois disto, é feito o *parse* da função, que produz os *tokens* que serão adicionados ao modelo interno do módulo, onde a informação do módulo é também atualizada.

O último passo consiste na inserção das variáveis locais na respetiva função. Tal como a criação da função, este passo é novamente equivalente à criação de variáveis globais. Apesar de ambas as implementações serem muito semelhantes, estas não só diferem no tipo da instrução que é inserida, mas também no contexto onde são inseridas, isto é, as variáveis locais são inseridas no contexto da função, ao contrário das variáveis globais que são inseridas no contexto do módulo. Para além disto, as variáveis globais são inicializadas na mesma instrução que as declara, enquanto que as variáveis locais são inicializadas através da criação de uma nova instrução que deve ser inserida no código da respetiva função.

Todo o processo relacionado com a criação de uma função, só é válido para os casos onde a função é interna ou declarada como sendo exportada. Esta última também exige a criação de um novo elemento do tipo *export* que deve apontar para a respetiva função. Para as funções importadas não existe o conceito de corpo da função, nem da presença de variáveis locais. Por essa razão, apenas é criada a instrução referente ao elemento *import*, que por sua vez declara a nova função, uma vez que contém na sua especificação a informação relativa aos seus parâmetros e valores de retorno.

Atenção, não é garantido que a interação com os elementos já existentes no módulo original (sem recurso ao contexto presente no ficheiro de transformações) funcione corretamente.

Caso o ficheiro de entrada tenha o formato textual, é quase certo que não funcionará, uma vez que este sofre algumas alterações antes de ser manipulado pela ferramenta (exceto se foi gerado pela ferramenta WABT (Smith & Community, 2021)). No caso de ter o formato binário, poderá funcionar no entanto é necessário que o utilizador tenha conhecimento do resultado que as ações definidas no ficheiro de transformação vão ter sobre o módulo original.

5.4.4. Transformação dos *Join-Points*

O primeiro passo desta fase baseia-se na execução e aplicação das *static expressions* presentes no código do *advice*. Para esta execução foram implementados um *lexer* e um *parser* internos. A separação entre *lexer* e o *parser* deve-se à dimensão da linguagem utilizada nas *static expressions*. Desta forma, não só simplifica a definição da gramática utilizada, como também não cria um acoplamento do *parser* com os detalhes lexicais da linguagem. Este desacoplamento foi fundamental para o processo de desenvolvimento da linguagem, que por sua vez foi evoluindo com o decorrer do projeto, sendo que os respetivos detalhes lexicais e sintáticos foram alterados diversas vezes durante o processo de desenvolvimento.

O *lexer* foi implementado baseado no algoritmo da apresentação *Lexical Scanning in Go* (Pike, 2011). Este é responsável por analisar o código, e à medida que progride na análise, emite os elementos que vai interpretando. O tipo de elementos emitidos varia de acordo com o contexto em que se encontram, e por isso, foi implementada uma máquina de estados. Esta permitia identificar o contexto que se encontrava a ser interpretado num dado momento, e assim, o *lexer* não só tinha a capacidade de perceber quais os tipos de elementos permitidos para um dado estado, como também os efeitos que esses elementos tinham na transição de estado.

Enquanto o *lexer* emitia elementos, o *parser* recebia-os e armazenava-os numa dada estrutura de dados interna. Todo este processo é feito de forma assíncrona, ou seja, o trabalho realizado pelo *lexer* é feito simultaneamente com o trabalho realizado pelo *parser*. Após a conclusão de ambos, a estrutura de dados deverá estar completamente preenchida e pronta a ser executada.

A execução da estrutura de dados resultante do *parse* do código é realizada com recurso a *Go Channels*, que foram implementados de forma a permitir uma comunicação bidirecional. Para isso, cada elemento da estrutura implementa um método comum que recebe como parâmetros um canal de saída (*emitter*) e um canal de entrada (*receiver*). Além disso, também recebem os "mapas" com os identificadores, uma vez que é necessário para fazer a substituição das referências utilizadas nas expressões. Com isto, é possível que os elementos presentes na estrutura recebam qualquer tipo de argumento, provenientes do último elemento que foi executado, e emitam qualquer tipo de resultado após a execução. Os tipos de dados disponíveis durante a execução destas expressões encontram-se especificados na Secção 3.1.3. Para fazer a leitura e escrita nos canais foi implementado um conjunto de estruturas próprias, que têm como objetivo receber e/ou emitir os dados de acordo com a sua finalidade.

Um exemplo deste tipo de estruturas, é o *receiver* chamado `TextOnlyReceiver` que lê dados de qualquer tipo, e converte-os para *string*. Foi também implementada uma estrutura deste tipo para cada uma das funções que pode ser aplicada aos identificadores.

Os identificadores que estão disponíveis para o acesso no código do *advice*, não só estão presentes no contexto global, como também no contexto local ao *advice*. Isto inclui, variáveis locais, parâmetros da função, dados de contexto gerados pela execução do *pointcut*, e as instruções provenientes do *join-point*. Apesar do contexto fornecido pela execução dos *pointcuts* ter sido feito no início do fluxo da transformação (Secção 5.4.1), os dados produzidos pelo *pointcut* `template` poderão sofrer algumas alterações. Isto porque quando são utilizadas múltiplas variáveis com o mesmo nome dentro do *template*, a ferramenta requer que todas tenham o mesmo valor, tornando-os assim sensíveis ao código específico do *join-point*. Com isto, os resultados que não respeitam esta condição são removidos do contexto.

O resultado final consiste num código WAT transformado, ou seja, com todas as *static expressions* resolvidas e convertidas no respetivo código. Este código é aplicado ao *join-point*, substituindo as instruções associadas ao *join-point*. É feito o *parse* deste código, e os *tokens* produzidos substituem todos os *tokens* que representam as devidas instruções na estrutura do módulo.

5.4.5. Transformação do Código das Novas Funções

Este processo foi separado da fase de criação das funções por causa dos seguintes motivos: podem interagir entre si, e assim determinadas referências podem não existir num dado momento da fase de criação; melhoria de desempenho, uma vez que a transformação para cada função foi implementada de forma concorrente; e por fim, pode existir a necessidade de haverem *join-points* que precisam de interagir e manipular variáveis utilizadas no código da função antes destas serem convertidas para o respetivo valor.

O modo de operação é muito semelhante à transformação realizada no código dos *join-points*, no entanto, os identificadores que estão disponíveis para o acesso nas novas funções são apenas aqueles que estão contidos no contexto global, o que inclui as variáveis globais e as funções adicionadas. Para além disso, também têm acesso aos dados que estão contidos no contexto da função, o que inclui as variáveis locais e os parâmetros da função. Por último, o código resultante da execução das *static expressions* é aplicado à função em questão apenas se existirem alterações. Todo este processo é executado paralelamente para cada uma das funções adicionadas.

5.4.6. Transformações *Runtime*

Existem três tipos de transformações *runtime*: as *runtime expressions*, as *runtime references*, e os tipos adicionais aos tipos existentes no WASM. Ao utilizar pelo menos um destes tipos de transformação, será sempre gerado um JS auxiliar ao módulo WASM. Como consequência, o utilizador terá de interagir obrigatoriamente com o JS para conseguir utilizar de forma correta o módulo WASM. Na Secção 4.4 encontra-se especificada a forma como esta interação deverá ser feita no lado do cliente.

Com isto, uma dessas classes é responsável pelo armazenamento das variáveis (classe `Zone`), tanto para o contexto das funções, como para o contexto global do módulo. Depois, existe uma classe para gerir os argumentos passados a cada função (classe `Args`), que se baseia na implementação de uma pilha, permitindo assim que cada função tenha o seu espaço de memória. Ainda relacionado com as funções, existe também uma classe para gerir o valores de retorno (classe `Returns`). Outra das classes existentes é a classe responsável pela execução das operações (classe `Operations`), onde está incluída a execução das expressões no lado do cliente. Por fim, há também uma classe usada para tratar erros no lado do cliente (classe `Error`).

Como foi mencionado acima, para existir a interação entre WASM e JS foram implementados um conjunto de métodos no objeto importado no módulo. Tais métodos servem para manipular a pilha dos argumentos, armazenar argumentos e valores de retorno, inicializar *scope* de variáveis para funções, armazenar variáveis locais e globais, e executar expressões. Para além destes métodos, de forma a permitir a inclusão de funções (importadas e exportadas) que recorrem ao uso dos tipos adicionais na sua declaração (parâmetros ou retorno), foi também implementada uma camada intermédia de métodos que faz toda essa gestão.

Cada transformação *runtime* utiliza estes métodos de forma distinta. Apesar disso, por regra, quando existe a necessidade de enviar valores para o lado do JS, caso o tipo do valor seja simples, isto é, valores do tipo *i32*, *f32* e *f64* (*i64* não é válido no JS), a sua transmissão mantém sempre o tipo, no entanto, caso o valor seja de um tipo complexo, isto é, *string*, *mapa* ou *array*, então é sempre enviada uma sequência de valores *i32*, que representa o respetivo valor num formato textual. Para isso, o JS deve ter conhecimento do tipo de dados que está a tratar, sendo necessário declarar sempre esse tipo antes de efetuar a transmissão.

A primeira subsecção (Secção 5.4.6.1) é responsável por detalhar a execução das *runtime expressions*, incluindo não só as modificações no módulo WASM, mas também as ações realizadas no JS. A próxima (Secção 5.4.6.2) vai tratar de uma forma breve as *runtime references*, e por fim, na última subsecção (Secção 5.4.6.3) vão ser abordados os novos tipos adicionados (tipos complexos) à linguagem, e o impacto vão causar no módulo.

5.4.6.1. Execução das Runtime Expressions

Para percorrer todas as expressões presentes no módulo, foi utilizado um *Visitor* que percorre o módulo processando cada uma das expressões. A execução destas expressões depende do tipo de instrução onde estão contidas, sendo que apenas são válidas chamadas a funções, alteração do valor das variáveis locais e globais, e operações que possuem na sua definição um tipo específico (por exemplo, a instrução `i32.const` espera como argumento um literal do tipo inteiro *32-bit*). Mais detalhes sobre as instruções disponíveis para o uso de *runtime expressions* encontram-se especificado na Secção 3.2.

A execução das expressões é sempre acompanhada por uma estrutura de dados auxiliar. Cada estrutura encontra-se associada a uma dada função, ou seja, as expressões contidas numa dada função partilham todas a mesma estrutura, e têm como objetivo garantir que a

execução é realizada com sucesso. Para isso, devem incluir quais as variáveis auxiliares que foram criadas durante o processo de execução das expressões e que precisam de ser inicializadas na função, devem garantir que a instanciação das variáveis simples no JS é realizada uma única vez (são instanciadas apenas para a realização da operação `eval`), e por fim, devem também fazer a gestão de toda o processo de criação de índices únicos na geração de código WAT.

Com isto, a execução começa por identificar a função a que a expressão pertence, e assim, obter a estrutura auxiliar necessária para a transformação. O próximo passo consiste na obtenção do tipo da instrução onde a expressão está inserida.

Caso a instrução seja uma chamada, a transformação dependerá do tipo do parâmetro da função que foi chamada. Se for do tipo simples, a transformação vai englobar a criação de uma variável local auxiliar, que será utilizada para armazenar o valor proveniente do resultado da expressão invocada no lado do JS. Depois de invocar a operação e armazenar o valor na variável auxiliar, a expressão, que neste momento consiste num argumento da chamada, será substituída pela instrução responsável por obter o valor da variável. Se for do tipo complexo, não é necessário qualquer variável auxiliar, uma vez que após invocar a operação, o resultado é copiado para o respetivo argumento no JS. No final, a expressão é removida da chamada.

Na Figura 63 está representado um pedaço de código WAT que consiste na atribuição do valor retornado pela chamada de uma dada função a uma variável local. Neste exemplo, a função tem como objetivo receber dois argumentos do tipo *string* e retornar a junção de ambos os argumentos separados por uma vírgula (valor de retorno é do tipo *string*). Como estas instruções englobam transformações *runtime*, serão representados os estados do contexto JS em cada um dos passos que ocorrem durante o processo. Estes passos são os seguintes:

0. Inicialmente, o estado do contexto JS está vazio (Figura 64).
1. Como será necessário interagir com variáveis locais complexas (*string*), será necessário criar um novo espaço de memória para as variáveis locais no início da função (Figura 65).
2. A chamada à função deve ser movida para o corpo da função, e na sua vez, deve ser adicionada uma *static expression* com a *keyword* `return_` de forma a obter o valor complexo devolvido pela chamada. Desta forma, este passo consiste na inicialização de uma nova *string* nas operações do JS (Figura 66).
3. O valor do primeiro argumento (“Hello”) é passado para o JS como um *array* de inteiros *i32* criado a partir de um conjunto de chamadas sucessivas a uma dada função (Figura 67).
4. Executar a operação `eval` no JS (Figura 68).
5. Apenas para o primeiro parâmetro, deve ser criado um novo espaço de memória para os respetivos argumentos de uma nova chamada (Figura 69).

6. O resultado obtido com a execução da operação `eval` é copiado para o respetivo argumento no JS (Figura 70).
7. Este resultado é depois limpo (Figura 71).
8. Para o segundo argumento, deve ser repetido o processo desde o passo 2. Contudo, no passo 3 invés de "Hello", deve ser passado o valor "world!", e o passo 5 deve ser completamente ignorado (Figura 72).
9. Executar chamada à função `join` que se encontra no módulo `operations` uma vez que possui tipos complexos na sua definição (Figura 73).
10. O espaço de memória anteriormente criado para os argumentos da chamada deve ser removido (Figura 74).
11. No seguimento da alteração explicada no passo 2 relativo ao valor de retorno através da *keyword* `return_`, deve ser inicializada uma nova *string* nas operações (Figura 75).
12. O valor "return_" é então passado, de forma análoga ao passo 3), como valor de entrada às operações no JS (Figura 76).
13. Executar operação `eval` no JS, utilizando o valor de retorno presente no JS (Figura 77).
14. Para atribuir o resultado da operação à respetiva variável local, é necessário inicializar uma nova *string* como uma variável local pendente (Figura 78).
15. Preencher o nome da variável a criar, de forma análoga ao passo 3 (Figura 79).
16. O resultado obtido com a execução da operação `eval` é depois posto na variável a criar (Figura 80).
17. Para concluir a criação da variável, é necessário proceder à sua submissão no JS, onde é realizado uma conjunto de operações tais como, verificação se já existe alguma variável com o nome criado e proceder à sua modificação, validação do tipo, etc. (Figura 81).
18. Novamente, o resultado nas operações deve ser limpo (Figura 82).
19. Mais tarde, no final da função, o espaço de memória para as variáveis locais deve ser destruído (Figura 83).

```
(local.set %result%
    (call %join% /"Hello"/ /"world!"/))
```

Figura 63 - Código WAT para exemplo com a função *join*

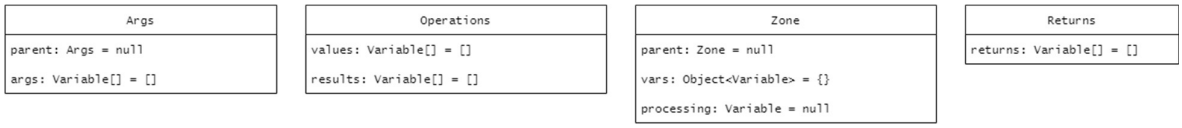


Figura 64 - Contexto incial do JS para o exemplo com a função *join*

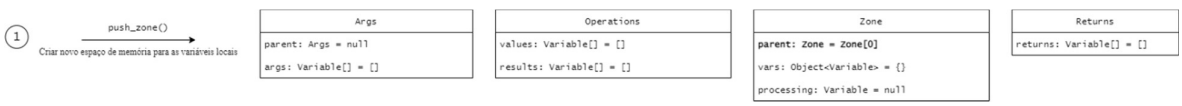


Figura 65 - Contexto do JS no 1º passo do exemplo com a função *join*

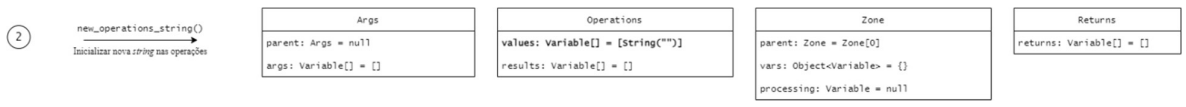


Figura 66 - Contexto do JS no 2º passo do exemplo com a função *join*

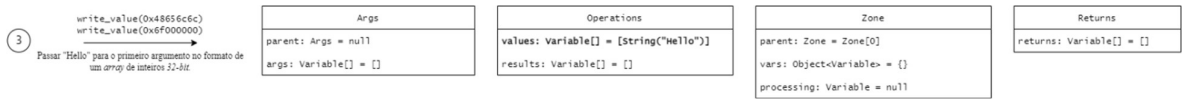


Figura 67 - Contexto do JS no 3º passo do exemplo com a função *join*

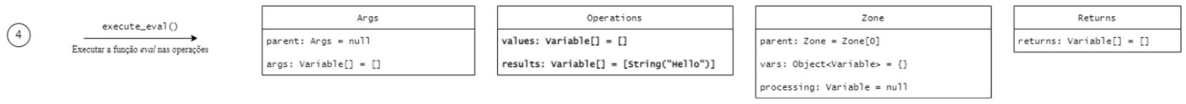


Figura 68 - Contexto do JS no 4º passo do exemplo com a função *join*

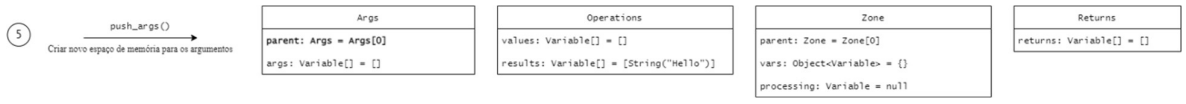


Figura 69 - Contexto do JS no 5º passo do exemplo com a função *join*

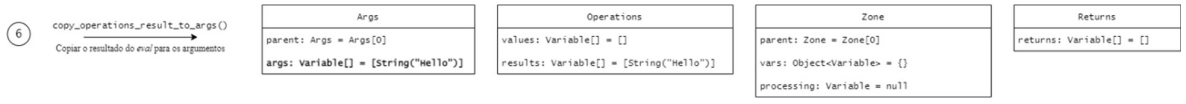


Figura 70 - Contexto do JS no 6º passo do exemplo com a função *join*

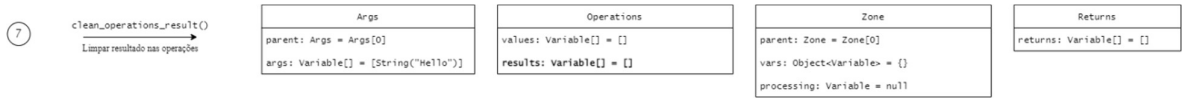


Figura 71 - Contexto do JS no 7º passo do exemplo com a função *join*

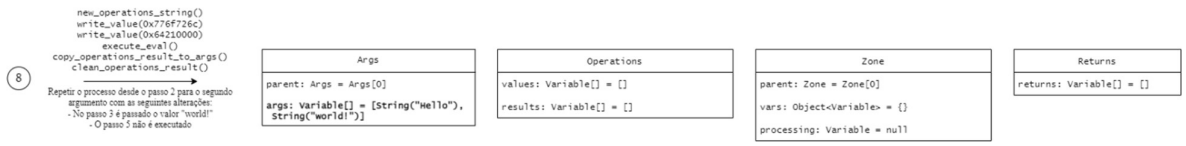


Figura 72 - Contexto do JS no 8º passo do exemplo com a função *join*

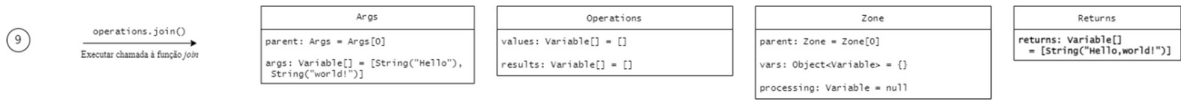


Figura 73 - Contexto do JS no 9º passo do exemplo com a função *join*

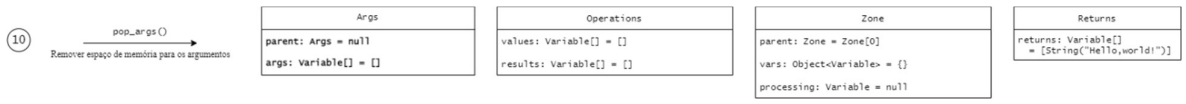


Figura 74 - Contexto do JS no 10º passo do exemplo com a função *join*

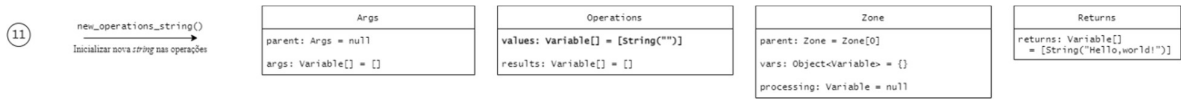


Figura 75 - Contexto do JS no 11º passo do exemplo com a função *join*

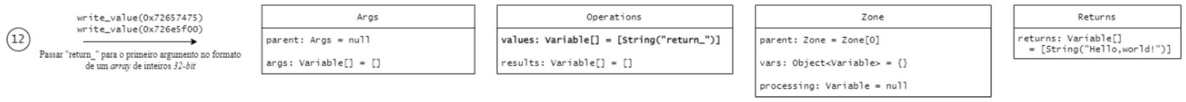


Figura 76 - Contexto do JS no 12º passo do exemplo com a função *join*

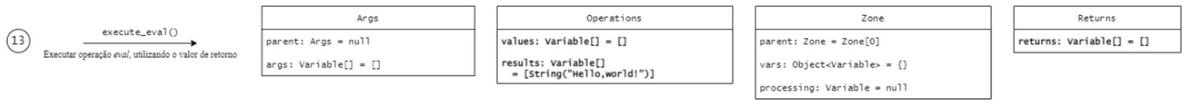


Figura 77 - Contexto do JS no 13º passo do exemplo com a função *join*

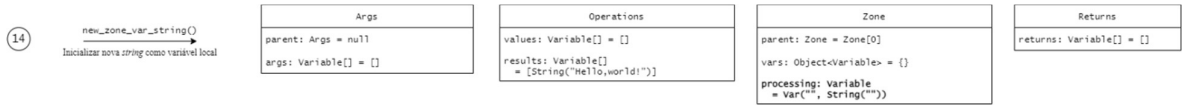


Figura 78 - Contexto do JS no 14º passo do exemplo com a função *join*

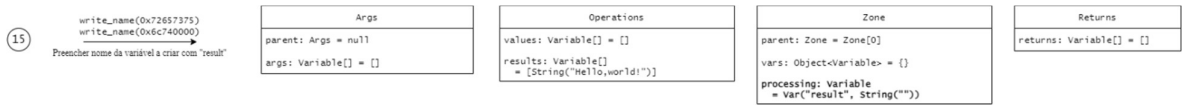


Figura 79 - Contexto do JS no 15º passo do exemplo com a função *join*

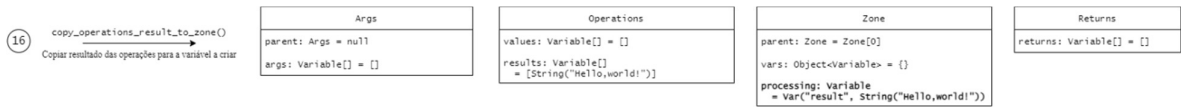


Figura 80 - Contexto do JS no 16º passo do exemplo com a função *join*

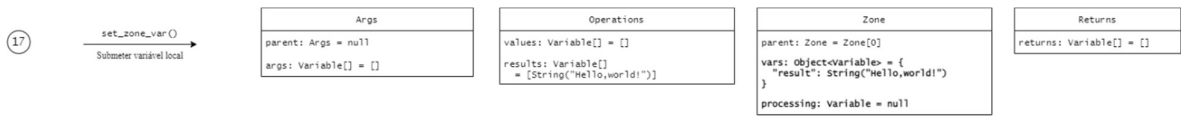


Figura 81 - Contexto do JS no 17º passo do exemplo com a função *join*

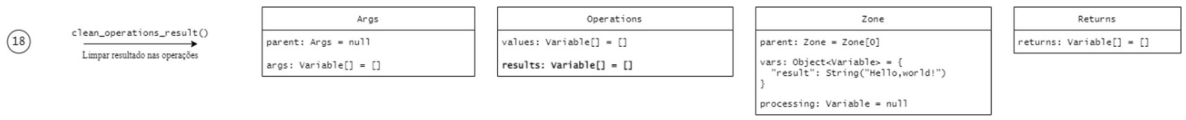


Figura 82 - Contexto do JS no 18º passo do exemplo com a função *join*

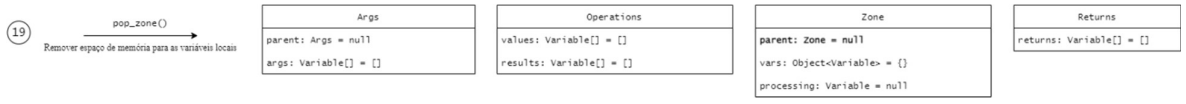


Figura 83 - Contexto do JS no 19º passo do exemplo com a função *join*

Os parâmetros do tipo complexo são sempre removidos da chamada, logo, neste exemplo tanto serão removidos os parâmetros na declaração da função, como os argumentos na instrução referente à chamada desta função.

As funções com o tipo de retorno complexo exigem transformações adicionais. As transformações englobam não só as instruções de retorno da função, mas também as instruções de chamada destas funções, tendo como objetivo armazenar o resultado no contexto do JS, e na chamada, substituir a chamada por uma *runtime expression* com a

keyword `return_` (permite aceder ao valor de retorno no lado do JS). Na Figura 73 é possível verificar este cenário, onde o valor de retorno é inserido na "pilha" de resultados da classe `Returns`.

Caso a instrução seja do tipo `local.set/tee` ou `global.set`, a transformação irá depender do tipo da variável que está a ser alvo de modificação. O processamento de variáveis do tipo simples é praticamente igual ao processamento descrito acima para os argumentos do tipo simples, sendo que a única diferença é que para além de alterar a expressão, é feita a verificação se a variável alvo está referenciada como uma *runtime reference*. Caso se verifique, essa referência é substituída pelo respetivo índice da variável. Se a variável for do tipo complexo, a transformação inclui a invocação da operação `eval` no JS, onde é feita a cópia do resultado obtido para a respetiva variável existente no contexto do JS.

As restantes instruções onde podem ser utilizadas *runtime expressions* aceitam apenas valores do tipo simples, e por isso, as transformações efetuadas são feitas de forma equivalente às transformações efetuadas para os argumentos do tipo simples mencionadas acima. Resumindo-se assim à criação de uma variável local auxiliar, que armazena o valor que resulta da invocação do `eval` no JS, e que substituirá a expressão na instrução (por um `local.get`).

5.4.6.2. Execução das Runtime References

As *runtime references* têm três objetivos principais: atribuir valores obtidos em tempo de execução a variáveis declaradas no ficheiro de transformações, permitir retornar variáveis complexas, e referenciar variáveis dentro das *runtime expressions*.

Para o primeiro, as *runtime references* estão diretamente relacionadas com as *runtime expressions* uma vez que estas referências apenas devem ser utilizadas em instruções do tipo `local.set/tee` e `local.get` que tenham como segundo argumento (valor que será atribuído à variável) uma *runtime expression*. Existiu a necessidade de implementar este meio de referenciar uma variável, uma vez que variáveis do tipo mapa e *array* permitem o acesso e manipulação aos seus membros, sendo algo que não é resolvido com a utilização dos índices das respetivas variáveis, mas sim através do JS. Desta forma, foi implementado um tipo de dados específico no contexto JS que permite o acesso aos membros de uma variável.

Relativamente ao segundo objetivo, foi necessário recorrer ao uso de *runtime references* principalmente para que fosse possível detetar quais as variáveis que estavam a ser devolvidas. Adicionalmente, também está relacionado com o primeiro objetivo, uma vez que em parte também serve para que seja possível devolver membros de variáveis. Com isto, a ferramenta consegue proceder às devidas transformações.

Finalmente, para o terceiro objetivo as referências vão permitir que a ferramenta consiga identificar quais as variáveis que devem ser substituídas pelo respetivo valor em tempo de compilação. Assim, a ferramenta insere todo o código necessário para realizar a sua gestão no lado do JS.

5.4.6.3. Tipos Complexos

Os tipos adicionais aos que existem no WASM, e que podem ser utilizados com recurso à ferramenta são: *strings*, mapas e *arrays*. Estes tipos, também conhecidos por tipos complexos, e apesar de serem declarados no ficheiro de transformações, não estão presentes no módulo WASM resultante da execução da ferramenta, uma vez que a linguagem WASM em si, neste momento, só suporta tipos simples. Desta forma, para que fosse possível utilizar tipos complexos, foi necessário arranjar uma alternativa que se baseou na utilização do ficheiro auxiliar JS.

O código presente no JS é responsável por armazenar todo o contexto associado às variáveis deste tipo, e para isso, todas as declarações e inicializações de elementos deste tipo necessitam de sofrer alterações. Isto engloba não só a declaração e inicialização das variáveis locais e globais, como também, a alteração dos parâmetros e retorno das funções.

Em relação às variáveis globais, foram criados dois *Visitors* com o intuito de procurar, ou em caso de não existir, criar uma função inicial no módulo, onde é incluído o código WASM que permite a declaração/inicialização destas variáveis no JS. Para as variáveis locais, durante o processo de execução das *runtime expressions*, vai sendo adicionado o código WASM que permite a declaração/inicialização dessas variáveis no JS.

Para as funções, tanto os parâmetro como o valor de retorno foi tratado não só durante a execução das *runtime references*, mas também numa fase posterior onde foi necessário implementar código JS que visa criar uma camada intermédia para as funções importadas e exportadas, e assim permitir fazer a gestão de parâmetros e valores de retorno internamente.

Por fim, foi implementado um *Visitor* responsável por remover todos os elementos do tipo complexo que restaram das transformações que foram realizadas durante a execução da ferramenta.

5.5. Geração de Resultados

O último passo da ferramenta consiste na geração dos resultados da transformação. Neste passo, é gerado, de forma concorrente, o módulo WASM transformado e o módulo JS caso tenha sido feita qualquer transformação *runtime* ou utilizados tipos desconhecidos ao WASM. O caminho e nome dos ficheiros são configurados pelo utilizador (Secção 5.2.1).

O código do módulo é primeiramente gerado no formato WAT, sendo que depois, com recurso à ferramenta WABT, este é transformado para o correspondente binário (WASM). No que toca ao código JS, este é gerado para um formato JS extenso, e depois, com recurso à ferramenta Minify, o ficheiro é devidamente otimizado.

CAPÍTULO 6.
VALIDAÇÃO

De forma a validar a ferramenta decidiu-se aplicar as várias técnicas descritas nos capítulos acima através de exemplos. O objetivo será cobrir um conjunto de transformações correspondentes a linguagens deste tipo. Alguns dos casos de uso habituais das linguagens orientadas a objetos são a instrumentação do código, melhorias de desempenho, correções temporárias, extensão de funcionalidades, entre outros (Restivo, 2006) (Team O. , 2004). Para além disto, os exemplos serão baseados em situações do contexto real, e por isso, cada exemplo tem por base código publicado.

Para validar a ferramenta WasmManipulator, os exemplos apresentados irão conter uma série de transformações aplicadas de forma iterativa, onde cada uma possui um propósito bem definido que pretende simular situações do contexto real. Para além de ser fornecida uma descrição do caso de uso referente a cada transformação, também será apresentado o respetivo código implementado.

Os exemplos criados irão abordar os requisitos definidos na Secção 4.1. Na Tabela 7 encontra-se descrita a lista com as principais funcionalidades e técnicas que se pretende validar nas várias transformações realizadas para cada exemplo.

Tabela 7 - Resumo de funcionalidades e técnicas aplicadas nas transformações de cada exemplo

Exemplo	***Transformações***	***Funcionalidades e Técnicas***
Stooge Sorting	*Logging* básico e monitorização de desempenho.	• Variáveis globais • Novas funções o Internas e importadas • *Pointcuts* o *Before* e *after* o Parâmetros de entrada
	Alterar para algoritmo mais rápido.	• *Pointcuts* o *Around* o Parâmetros de entrada • *Order*
	Planos de subscrição.	• Inclusão de todas as alterações do primeiro exemplo
Filtros de Imagens	Filtro baseado na função seno.	• Novas Funções o Interação entre si o Maior complexidade
	Monitorização de desempenho.	• Semelhante ao exemplo anterior
	Caching.	• Memória linear • Variáveis globais
	Reparação de um erro.	• *Templates*

Exemplo	***Transformações***	***Funcionalidades e Técnicas***
	Otimizações *Sepia*.	• *Template* generalizado • Combinação de *templates* • *Static expressions* • *Runtime expressions* • Contexto do *pointcut*
	Melhoramentos com JS.	• Templates • Runtime expressions • Novos tipos de dados
Markdown Parser	Análise.	• *Static expressions* + *Runtime expressions*
	Implementação da contagem.	• Função inicial com inicialização de dados • *Runtime expressions* • *Runtime references* • Novos tipos de dados ○ *Around* ○ Parâmetros de entrada • *Order*
Jogo Invaders	Implementação dos novos modos.	• Pointcuts globais • Função inicial com configurações

Sabendo isto, o primeiro exemplo (Secção 6.1) consiste num programa utilizado para fazer a ordenação de um array de inteiros. A primeira transformação ao programa consistiu na inserção de um método que permitiu fazer a instrumentação da função principal de ordenação. Com esta transformação, pretendeu-se fazer o registo de quando o processo de ordenação é iniciado e quando é terminado. De seguida, foi implementado um método para permitir o registo do tempo que cada execução demora. Para isso, a função inserida na transformação anterior foi adaptada para que, além de fazer o registo na consola, também faça o devido controlo. Esta modificação confirmou que o algoritmo de ordenação utilizado pelo programa, o *Stooge Sort* (Team G. , StoogeSort, 2021), realmente é extremamente ineficiente. Desta forma, procedeu-se à substituição da função existente por um função que implemente um algoritmo mais eficiente, o *Heap Sort* (Team G. , HeapSort, 2021). Por fim, para simular um sistema de subscrições, foi implementada uma condição adicional ao programa que de acordo com o plano subscrito pelo utilizador, executava um destes métodos de ordenação.

O próximo exemplo (Secção 6.2) consiste num programa que permite a aplicação de filtros a imagens (Tulka, 2021). Os filtros disponibilizados pela aplicação são: "invert", "grayscale", "basic monochrome" e "random monochrome". Com isto decidiu-se criar um novo filtro chamado "sin" que permitia aplicar a função do seno (com recurso a uma outra fórmula) a cada pixel da imagem. Através da aplicação de um método de monitorização semelhante ao aplicado no exemplo anterior, confirmou-se que este novo filtro era mais pesado que os restantes filtros da aplicação, precisando assim de mais tempo para ser aplicado. Com isto, foi implementado um mecanismo de *cache* que recorria à memória linear

do módulo WASM para armazenar e devolver os respetivos valores associados a uma dada cor. Depois desta otimização, será alterada a expressão que calculava o valor da cor do pixel para incluir um erro que, dependendo da cor, fizesse com que a execução abortasse. Com isto, foi realizada a devida correção ao programa. De seguida, foi acrescentado o filtro *Sepia* à aplicação. Para além do filtro foi adicionado uma nova função que executa cem mil vezes o filtro *Sepia*. Esta função serve para testar de que modo é que a aplicação deste filtro alterava a escala de cores da imagem, e como previsto, a aplicação do filtro aclara a imagem uma vez que o resultado do método é uma imagem cujos píxeis possuem todos a cor branca. Para acelerar este processo foram implementadas duas técnicas de otimização, que consistem na remoção de instruções com chamadas a funções e na técnica de *Loop Unrolling*.

Até ao momento, no exemplo definido acima, os filtros foram aplicados individualmente e de forma sequencial. No entanto, se for necessário aplicar os filtros de forma concorrente, o processo de registo e monitorização da função irá falhar uma vez que não é possível identificar a função que invocou a ação. Desta forma, a função responsável por isto foi alterada para que incluísse num dos parâmetros o nome da função que a invocou (apenas para clientes JS).

Ainda no mesmo exemplo, até ao momento só se estava a fazer *cache* para o filtro "sin". Contudo, o filtro "invert", apesar de não ser tão pesado, também poderá beneficiar desta melhoria (os restantes filtros ou exigem uma implementação mais elaborada, ou não é possível implementar *cache*). Sendo assim, foi implementado um mecanismo de *cache* que permite ser aplicado a ambos os filtros (ou a qualquer outro filtro que permita a implementação de um mecanismo de *cache* semelhante). Tal como a transformação acima, esta é também suportada apenas por clientes JS.

Para o terceiro exemplo (Secção 6.3) foi implementada uma extensão a um programa destinado ao *parser* para Markdown (Andersson, 2021). Esta extensão consistiu na inserção da contagem de elementos existentes no texto. Com este exemplo, não só foi possível demonstrar que a ferramenta permite estender aplicações com funcionalidades relevantes para o utilizador, como também demonstrar a sua utilidade na análise de código WASM já publicado, uma vez que para a realização desta transformação foi necessário realizar uma análise deste código.

Por último, será apresentado um exemplo (Secção 6.4) com a inclusão de dois modos diferentes ao jogo Invaders. Estes modos são configuráveis pelo utilizador através das funções importadas que são executadas no início da execução do programa.

6.1. Stooge Sorting

O primeiro exemplo consiste num programa para ordenação de um *array* de inteiros *32-bit*. O algoritmo utilizado na implementação foi o *Stooge Sorting*, que por definição é um algoritmo de ordenação recursiva. Por natureza, o algoritmo oferece um mau desempenho uma vez que a sua complexidade é superior à complexidade quadrática ($O(N^{2.709})$). O código WAT foi gerado através da ferramenta “WebAssembly Explorer” (Bebenita, 2018) que

permitiu a conversão do código implementado na linguagem C obtido na página do "GeekforGeeks" (Team G. , StoogeSort, 2021). A execução do programa WASM foi realizada através do NodeJS.

6.1.1. Transformação – *Logging* básico

A primeira transformação realizada baseou-se na inserção de uma função que permitia o programa fazer o registo de um inteiro para a consola, sendo esta chamada no início e fim da execução da função de ordenação. Com isto, antes da execução da função é feito uma chamada à função de *logging* com o código "0" nos argumentos, e depois da execução, é feita a mesma chamada no entanto com o código "1". Para esta modificação foi necessário ter em atenção o facto da função de ordenação ser recursiva, e por isso foi necessário implementar um mecanismo que permitisse identificar esta recursividade.

Na Figura 84 encontra-se o código escrito na ferramenta para efetuar esta alteração, que englobou a criação de dois *advices*, duas novas funções e uma variável global.

```
templates:
  t: (local %_% i32) (local %_% i32) (local %_% i32) (local %_% i32)
aspects:
  context:
    functions:
      log_sort:
        args:
          - name: type
            type: i32
        imported:
          module: input
          field: log_sort
      log_sort_internal:
        args:
          - name: was_in
            type: i32
          - name: type
            type: i32
        code: >
          (block $B0
           (br_if $B0
            (i32.eqz
             (local.get %was_in%)
            )
           )
           (return)
          )
          (call %log_sort% (local.get %type%))
    variables:
      is_in: i32
  advices:
    sort_around_before:
      order: 0
      pointcut: () => func(void *(i32, i32, i32)) && template(t, true)
      variables:
        was_in: i32
      advice: >
        (local.set %was_in% (global.get %is_in%))
```

```
        (global.set %is_in% (i32.const 1))
        (call %log_sort_internal% (local.get %was_in%) (i32.const 0))
        %this%
    sort_around_after:
      order: 1
      pointcut: (i32.local[was_in] was_in) => func(void *(i32, i32, i32)) && te
mplate(t, true) && returns(void)
      advice: >
        (call %log_sort_internal% (local.get %was_in%) (i32.const 1))
        (global.set %is_in% (local.get %was_in%))
```

Figura 84 - Código da transformação para o *logging* básico da função de ordenação

Relativamente aos *advices*, o primeiro serviu para acrescentar a chamada à função de *logging* no início da função de ordenação, e o segundo também tinha como função acrescentar uma chamada à função de *logging*, contudo, no final da função de ordenação. A definição dos *pointcuts* existentes em ambos os *advices* é muito semelhante, uma vez que a função que se pretende alterar é a mesma. Ambos utilizam uma combinação das funções `func` e `template` que textualmente pode ser traduzido para o seguinte: os *join-points* devem pertencer a uma função que recebe três argumentos do tipo inteiro *32-bit* e não devolve nenhum valor (`func(void *(i32, i32, i32))`), e também deve declarar quatros variáveis do tipo inteiro *32-bit* (`template(t, true)`, em que `t` é o template com essa definição). A execução destes *pointcuts* resultará na obtenção de um *join-point* associado à função de ordenação que contém todas as instruções presentes no seu "corpo". No primeiro *advice*, para que fosse simulado o comportamento de um "*before*" (comum nas linguagens AOP), foi adicionado o identificador `this` após a chamada ao *logging*. Para o caso do *log* no final da função, esta chamada não seguiu a mesma abordagem, ou seja, não foi colocada logo após a instrução com o identificador `this`, uma vez que a função poderá retornar antes desta instrução. Para simular o comportamento do "*after*" (comum nas linguagens AOP), foi necessário adicionar um *advice* com a função `returns`, uma vez que este visa obter todas as instruções de retorno existentes numa dada função. Desta forma, foi adicionada esta função na expressão do *pointcut* do segundo *advice*, e como consequência, para além de procurar a função de ordenação, este obtém todos os *join-points* associados às várias instruções de retorno, adicionando a chamada antes de cada uma dessas instruções.

Foi adicionada uma função importada que contém a implementação do serviço de *logging* no lado do cliente. Depois também foi adicionada uma função que é invocada no programa, e serve de fachada para a função importada. Esta função foi implementada para facilitar o mecanismo de proteção contra a recursividade existente na função de ordenação. Como é possível visualizar na Figura 84 foi inserida uma variável no primeiro *advice* executado pela ferramenta, que por sua vez também é inserida no segundo *advice* através dos parâmetros do *pointcut*. Esta variável, juntamente com a variável global, serve para perceber se a função está a ser executada pelo utilizador ou de forma recursiva. O valor da variável é passada à função, que de acordo com o seu estado, chama a função importada ou não.

O código JS para a função importada encontra-se ilustrado na Figura 85. Esta foi importada com o nome `log_sort` para o módulo `env`, e tem como objetivo fazer o registo do estado para

a consola. O estado da consola após a execução do programa com esta transformação pode ser observado na Figura 86.

```
// handles the log emitted by the `sort` function
function handleLogSort(code) {
    // log to the console
    console.log(`Running state: ${code}`);
}

// ...

// import object for the WASM module
const importObject = {
    env: {
        log_sort: (code) => handleLogSort(code)
    }
};

// create the WASM instance
const wasm = await WebAssembly.compile(fs.readFileSync('./output.wasm'));
const instance = await WebAssembly.instantiate(wasm, importObject);

// ...
```

Figura 85 - Código JS para o *logging* básico da função de ordenação

```
Output:
> `Running state: 0`
> `Running state: 1`
> `Int32Array [ 0, 5, 8, 9, 10, 11, 11, 13, 16, 19 ]`
```

Figura 86 - Resultado na consola da transformação para o *logging* básico da função de ordenação

6.1.2. Transformação – Monitorização de Desempenho

Com recurso ao método adicionado no passo anterior, que tem como função fazer o registo do estado da execução da função de ordenação, foi implementado no lado do JS uma forma de contabilizar o tempo total que demora cada execução dessa função (Figura 87). O tempo necessário para fazer a ordenação de um *array* de dois mil elementos utilizando a função associada ao algoritmo Stooge Sorting encontra-se expresso na Figura 88, que por sua vez contém o estado da consola após a execução do programa.

```
// handles the log emitted by the `sort` function
function handleLogSort(code) {
    console.log(`Running state: ${code}`);    // log to the console
    if (code === 0) {
        console.time("sort");    // record starting function time
        return;
    }
    console.timeEnd("sort");    // calculates the `sort` duration
and prints it to the console
}
```

Figura 87 - Código da transformação para a monitorização de desempenho da função de ordenação

```
Output:
> `Running state: 0`
> `Running state: 1`
> `sort: 4823.935ms`
> `Int32Array [ 1, 1, 1, 1, 2, 2, 2, 3, 3, 4 ]`
```

Figura 88 - Resultado na consola da transformação para a monitorização de desempenho da função de ordenação

6.1.3. Transformação – Alterar para Algoritmo mais Rápido

Segue-se a transformação que visa substituir o algoritmo de ordenação presente no programa por um algoritmo com melhor desempenho. O novo algoritmo é conhecido por *Heap Sort*, e consiste num algoritmo cuja complexidade é O(n log n), logo é muito mais eficiente que o anterior. Este implementa uma técnica de classificação baseada em comparação com base na estrutura de dados *Binary Heap*. Tal como foi feito para o algoritmo anterior, utilizou-se uma ferramenta para converter o código C obtido no "GeekforGeeks" (Team G. , HeapSort, 2021), e depois de obter o código WAT foram feitas as devidas alterações para este fosse integrado no programa.

Esta alteração envolveu a criação de um novo *advice* que tem como objetivo a substituição da função de ordenação, e a inserção de uma nova função utilitária pertencente ao algoritmo *Heap Sort*. Uma vez que o alvo do *advice* é o mesmo que o que insere uma chamada ao *logger* no início da função de ordenação, ambos os *advices* possuem a mesma expressão no *pointcut* (explicada acima). A diferença deste *advice* é que, enquanto que a chamada ao *logger* simula o comportamento de um "*before*", este simula o comportamento de um "*around*" (comum nas linguagens AOP). Desta forma, como não foi utilizado o identificador `this` para inserir o código da função, todas as instruções anteriormente existentes vão ser substituídas pelas novas instruções expressas no *advice*. Estas instruções correspondem à implementação do novo algoritmo, e como esta implementação necessita de aceder aos parâmetros da função, foi necessário incluir cada um dos parâmetros no contexto do *pointcut* (`(i32.param[0] arr, i32.param[1] start, i32.param[2] end) => ...`). Os *advices* definidos no passo anterior mantêm-se, no entanto, são executados apenas após a substituição da função de ordenação, sendo realizadas sobre a nova implementação. A Figura 89 ilustra um excerto do código para a realização da transformação.

```
(...)
aspects:
  context:
    functions:
      (...)
      heapify:
        args:
          - name: arr
            type: i32
          - name: n
            type: i32
          - name: i
            type: i32
```

```
        variables:
          v1: i32
          v2: i32
          v3: i32
        code: >
          (...) ;; implementação
    (...)
  advices:
    replace_sort:
      order: 0
      pointcut: (i32.param[0] arr, i32.param[1] start, i32.param[2] end) => fun
c(void *(i32, i32, i32)) && template(t, true)
      variables:
        v1: i32
        v2: i32
        v3: i32
      advice: >
        (block $label$0
         (br_if $label$0
          (i32.lt_s
           (tee_local %v3%
            (i32.add
             (i32.sub
              (i32.const 1)
              (get_local %start%)
             )
             (get_local %end%)
            )
           )
           (i32.const 2)
          )
         )
         (set_local %v1%
          (i32.add
           (get_local %end%)
           (i32.const 1)
          )
         )
         (set_local %v3%
          (i32.add
           (i32.add
            (get_local %start%)
            (i32.shr_u
             (get_local %v3%)
             (i32.const 1)
            )
           )
           (i32.const -1)
          )
         )
         (loop $label$1
          (call %heapify%
           (get_local %arr%)
           (get_local %v1%)
           (get_local %v3%)
          )
          (set_local %v2%
           (i32.gt_s
            (get_local %v3%)
```

```
            (get_local %start%)
           )
          )
          (set_local %v3%
           (i32.add
            (get_local %v3%)
            (i32.const -1)
           )
          )
          (br_if $label$1
           (get_local %v2%)
          )
         )
        )
        (...) ;; resto da implementação
    sort_around_before:
      order: 1
      pointcut: () => func(void *(i32, i32, i32)) && template(t, true)
      variables:
        was_in: i32
      advice: >
        (local.set %was_in% (global.get %is_in%))
        (global.set %is_in% (i32.const 1))
        (call %log_sort_internal% (local.get %was_in%) (i32.const 0))
        %this%
    sort_around_after:
      order: 2
      pointcut: (i32.local[was_in] was_in) => func(void *(i32, i32, i32)) && te
mplate(t, true) && returns(void)
      advice: >
        (call %log_sort_internal% (local.get %was_in%) (i32.const 1))
        (global.set %is_in% (local.get %was_in%))
```

Figura 89 - Código da transformação para um algoritmo de ordenação mais rápido

```
Output:
> `Running state: 0`
> `Running state: 1`
> `sort: 0.374ms`
> `Int32Array [ 0, 0, 0, 2, 3, 5, 5, 6, 6, 6 ]`
```

Figura 90 - Resultado na consola da transformação para um algoritmo de ordenação mais rápido

6.1.4. Transformação – Planos de Subscrição

Segue-se uma transformação que visa executar dois algoritmos diferentes de acordo com um estado indicado pelo cliente. Com isto, é exemplificado um caso onde um utilizador poderá ter diferentes regalias de acordo com o plano subscrito. Para aplicar a transformação foi adicionada uma função importada (`pro_version`) que devolve "1" se o utilizador possui um plano que lhe permite executar o algoritmo com melhor desempenho (*Heap Sort*), e "0" se o utilizador não possui esse plano. Para isso, ambas as implementações dos algoritmos foram colocados em funções isoladas. Depois disso, foi modificado o *advice* anteriormente definido, para que segundo o estado retornado pela função `pro_version`, invoque uma das

funções de ordenação. O código necessário à transformação encontra-se ilustrado na Figura 91.

```
(...)
aspects:
  context:
    functions:
      (...)
      pro_version:
        result: i32
        imported:
          module: env
          field: pro_version
      heap_sort:
        args:
          - name: arr
            type: i32
          - name: start
            type: i32
          - name: end
            type: i32
        variables:
          v1: i32
          v2: i32
          v3: i32
        code: >
          (...) ;; implementação
      (...)
    (...)
  advices:
    toggle_sort:
      order: 0
      pointcut: (i32.param[0] arr, i32.param[1] start, i32.param[2] end) => fun
c(void *(i32, i32, i32)) && template(t, true)
      advice: >
        (block $label$0
         (br_if $label$0
          (call %pro_version%)
         )
         %this%
         (return)
        )
        (call %heap_sort%
          (local.get %arr%)
          (local.get %start%)
          (local.get %end%)
        )
    (...)
```

Figura 91 - Código da transformação com o plano de subscrição para a função de ordenação

O código JS para a função importada encontra-se ilustrado na Figura 92, sendo esta função foi adicionada ao módulo `env`. O resultado da execução do programa modificado para um utilizador sem o melhor plano está ilustrado na Figura 93, enquanto que para um utilizador com este plano está ilustrado na Figura 94. Como é possível observar, o desempenho obtido

pelo utilizador com o melhor plano é muito superior ao desempenho obtido pelo outro utilizador.

```
// ...

// import object for the WASM module
const importObject = {
    env: {
        log_sort: (code) => handleLogSort(code),
        pro_version: () => isUserVersionPro() // Returns 1 if true
    }
};

// ...
```

Figura 92 - Código JS para a transformação com o plano de subscrição para a função de ordenação

```
Output 1:
> `Running state: 0`
> `Running state: 1`
> `sort: 5377.452ms`
> `Int32Array [ 0, 2, 2, 2, 4, 5, 6, 8, 9, 9 ]`
```

Figura 93 - Resultado na consola da execução da função de ordenação sem o melhor plano

```
Output 2:
> `Running state: 0`
> `Running state: 1`
> `sort: 0.436ms`
> `Int32Array [ 1, 2, 3, 3, 3, 3, 4, 4, 4, 5 ]`
```

Figura 94 - Resultado na consola da execução da função de ordenação com o melhor plano

Este exemplo é meramente informativo, sendo que a sua aplicação não deverá ser feita no lado do cliente uma vez que este vai ter acesso ao código gerado e poderá manipulá-lo para seu benefício. Contudo, existem diversas alternativas para que seja feito no lado do servidor, tais como criar um *advice* que invalida todas as secções do código a que o utilizador não tem direito, ou mais simples, criar um *advice* para cada plano, e executar apenas o *advice* referente ao plano (configuração da execução).

6.2. Filtros de Imagens

O próximo exemplo consiste num programa utilizado para aplicar filtros em imagens. Este programa foi desenvolvido por Thomas Tulka e encontra-se publicado no GitHub (Tulka, 2021). Neste programa encontram-se implementados quatro filtros: "invert", "grayscale", "basic monochrome" e "random monochrome". Para o exemplo vai ser implementada uma extensão às funcionalidades do programa, onde serão adicionados os filtros "sin" e "sepia". Para além disso, vão ser feitas algumas otimizações como *caching* e *loop unrolling*. Estas otimizações vão ser testadas e comparadas após a inserção de um mecanismo de monitorização de desempenho no programa. No final vão ser acrescentadas algumas

modificações que englobam a integração com o módulo auxiliar JS, que inclui, a monitorização e registo de múltiplas funções e a implementação do mecanismo de *cache* universal.

Para testar as transformações presentes neste exemplo foram utilizadas duas imagens com dimensão *4320x7680* píxeis. Estas imagens encontram-se ilustradas na Figura 95, sendo retiradas do Wallpaper Flare (Community W. F., 2021). A primeira imagem foi renomeada de “picture1.jpg” e a segunda de “picture2.jpg”.

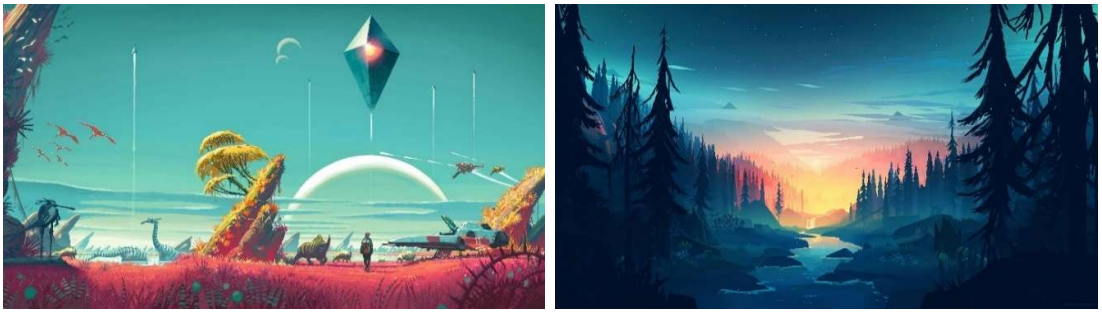


Figura 95 - Imagens de teste na aplicação de filtros de imagens ("picture1.jpg" e "picture2.jpg”)

6.2.1. Transformação – Filtro Baseado na Função Seno

A primeira transformação realizada sobre o código do programa consiste na inserção de um novo filtro baseado na função do seno, que tem como objetivo calcular o valor da cor para cada pixel da imagem com recurso à expressão $\min(255, abs(\frac{7}{3} sin\ x))$. Para isso foram implementadas as funções para calcular o seno de um valor, e para aplicar a cor calculada para cada um dos píxeis da imagem. As funções para calcular o seno foram implementadas com base nas séries de Taylor (Math, 2012). Esta implementação foi realizada na linguagem C e foram convertidas em WAT através da mesma ferramenta utilizada no exemplo anterior, o "WebAssembly Explorer", que depois foi modificado para ser incluído na ferramenta. Em relação às funções que têm como objetivo a aplicação do seno sobre os píxeis da imagem, estas foram implementadas inteiramente em WAT, usando como guia as funções já existentes no código para os restantes filtros. Na Figura 96 encontra-se representado o código completo para a aplicação desta transformação.

```
aspects:
  context:
    functions:
      factorial:
        args:
          - name: n
            type: f64
        result: f64
        code: >
          (...) ;; implementação
      power:
        args:
          - name: base
            type: f64
          - name: exp
```

```
        type: f64
    result: f64
    variables:
      (...)
    code: >
      (...) ;; implementação
  fmod:
    args:
      - name: x
        type: f64
      - name: y
        type: f64
    result: f64
    variables:
      (...)
    code: >
      (...) ;; implementação
  sin:
    args:
      - name: x
        type: f64
    result: f64
    variables:
      v1: f64
      v2: f64
      v3: i32
      v4: i32
    code: >
      (...) ;; implementação
  color_sin:
    args:
      - name: x
        type: i32
    result: i32
    code: >
      (i32.trunc_f64_u
       (f64.min
        (f64.mul
         (f64.abs
          (call %sin%
           (f64.convert_i32_u
            (local.get %x%)
           )
          )
         )
         (f64.mul
          (f64.convert_i32_u
           (local.get %x%)
          )
          (f64.const 2.333333333)
         )
        )
        (f64.const 255)
       )
      )
  sin_modification:
    exported: sin
    args:
      - name: width
```

```
        type: i32
      - name: height
        type: i32
    variables:
      i: i32
    code: >
      (local.set %width%
       (i32.shl
        (i32.mul
         (local.get %width%)
          (local.get %height%))
        (i32.const 2)))
      (loop $L0
       (if $I1
        (i32.gt_s
         (local.get %width%)
         (local.get %i%))
        (then
         (i32.store8
          (local.get %i%)
          (call %color_sin%
            (i32.load8_u (local.get %i%))
          ))
         (i32.store8 offset=1
          (local.get %i%)
          (call %color_sin%
            (i32.load8_u (i32.add (local.get %i%) (i32.const 1)))
          ))
         (i32.store8 offset=2
          (local.get %i%)
          (call %color_sin%
            (i32.load8_u (i32.add (local.get %i%) (i32.const 2)))
          ))
         (local.set %i%
          (i32.add
           (local.get %i%)
           (i32.const 4)))
         (br $L0))))
```

Figura 96 - Código da transformação que adiciona o filtro “sin”

Para executar o programa foi implementada uma aplicação *web*, que teve por base o código existente no programa original. Para esta aplicação foi criado um ficheiro HTML que contém um elemento *canvas* onde será desenhada a imagem. O código JS (Figura 97) executa o programa e "desenha" o resultado neste elemento. Na Figura 98 encontra-se um exemplo de aplicação do novo filtro nas duas imagens de teste com dimensões de *4320x7680* píxeis.

```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext('2d');
const [width, height] = [canvas.width, canvas.height];

const img = new Image();
img.src = './picture1.jpg';
img.crossOrigin = 'anonymous';
img.onload = () => transform();
```

```
// transforms the image applying the `sin` filter
async function transform() {
    // initialize WASM import data
    const arraySize = (width * height * 4) >>> 0;
    const nPages = ((arraySize + 0xffff) & ~0xffff) >>> 16;
    const memory = new WebAssembly.Memory({ initial: nPages });
    const importObject = {
        env: {
            memory,
            abort: (_msg, _file, line, column) => console.error(`Abort at ${lin
e}:${column}`),
            seed: () => new Date().getTime()
        }
    };

    // create the WASM instance
    const response = await fetch('./output.wasm');
    const responseBuffer = await response.arrayBuffer();
    const { instance } = await WebAssembly.instantiate(responseBuffer, importOb
ject);
    const wasm = instance.exports;

    // get the original image data
    const imageData = originalImageData();

    // get the output image buffer
    // and copy the original image data to it.
    const bytes = new Uint8ClampedArray(wasm.memory.buffer);
    copyData(imageData.data, bytes);

    // perform `sin` transformation
    // to change the output image buffer
    instance.exports.sin(width, height, cachePtr);

    // write the transformed image data to the canvas
    writeImageData(imageData, bytes);
}

// cleans the canvas with the original data
// and returns its data array
function originalImageData() {
    // draw the original image
    // and return the image data
    ctx.drawImage(img, 0, 0, width, height);
    return ctx.getImageData(0, 0, width, height);
}

// copies the data from `src` array to `dest`
function copyData(src, dest) {
    // copy each element, one by one
    for (let i = 0; i < src.length; i++) {
        dest[i] = src[i];
    }
}

// writes some data array to the image data
// and draws it to the canvas
function writeImageData(imageData, bytes) {
    const data = imageData.data;
```

```
    // copy each element, one by one
    for (let i = 0; i < data.length; i++) {
        data[i] = bytes[i];
    }
    // draw the image data to the canvas
    ctx.putImageData(imageData, 0, 0);
}
```

Figura 97 - Código JS para a execução do filtro “sin” após transformação GitHub (Tulka, 2021)

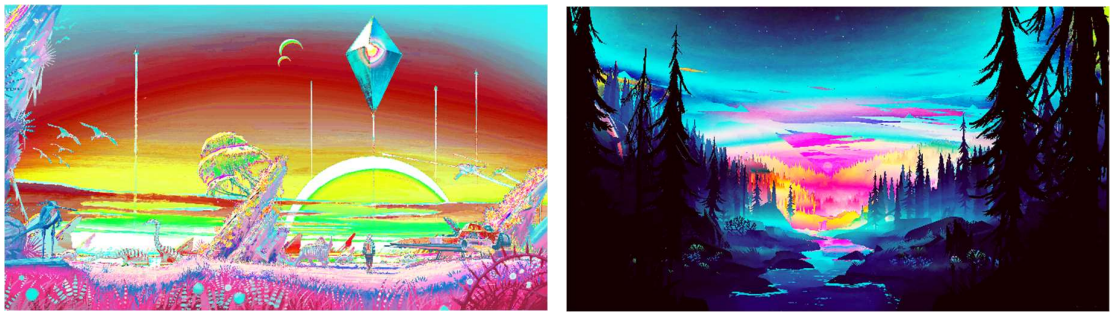


Figura 98 - Resultado da execução do filtro “sin” sobre as imagens de teste "picture1.jpg" e "picture2.jpg”

6.2.2. Transformação – Monitorização de Desempenho

Tal como no exemplo anterior, foi implementada uma alteração com o intuito de contabilizar o tempo total que demorou a execução de uma determinada função, que neste caso, para contabilizar o tempo leva um determinado filtro a ser aplicado. Com isto, o processo de aplicação das transformações é equivalente, e desta forma também foram adicionados dois *advices* com o objetivo de encontrar as funções onde se pretende adicionar a chamada à função que faz a monitorização dos tempos de execução, antes e depois. A expressão comum em ambos os *pointcuts* pode ser traduzida textualmente para o seguinte: os *join-points* devem pertencer a uma função com o nome "sin" ou uma função exportada que não devolve nenhum valor (`func(void sin(..)) || func(void *(..), exported)`), e para cada um dos casos, deve escrever valores para o *array* com os píxeis da imagem (`template(t, true)`, em que `t` é o template com essa definição). O código implementado para a transformação pode ser visualizado na Figura 99.

```
templates:
  t: (i32.store%_%)
aspects:
  context:
    functions:
      monitoring:
        args:
          - name: type
            type: i32
        imported:
          module: env
          field: monitoring
      (...)
  advices:
    monitor_around_start:
      all: true
```

```
      order: 0
      pointcut: () => func(void sin(..)) || func(void *(..), exported) && templ
ate(t, true)
      advice: >
        (call %monitoring% (i32.const 0))
        %this%
    monitor_end:
      all: true
      order: 1
      pointcut: () => (func(void sin(..)) || func(void *(..), exported) && temp
late(t, true)) && returns(void)
      advice: >
        (call %monitoring% (i32.const 1))
        %this%
```

Figura 99 - Código da transformação para a monitorização de desempenho da função para o filtro “sin”

O código adicionado no JS relativo à importação da nova função de monitorização encontra-se ilustrado na Figura 100. Na Figura 101 encontra-se o resultado obtido na consola do navegador após a execução do programa modificado para ambas as imagens de teste utilizadas anteriormente.

```
// (...)

// transforms the image applying the `sin` filter
async function transform() {
  // (...)
  const importObject = {
        env: {
            memory,
            abort: (_msg, _file, line, column) => console.error(`Abort at ${lin
e}:${column}`),
            seed: () => new Date().getTime(),
            monitoring: (code) => monitorFilter(code)
        }
    };
    // (...)
}

// (...)

// handles the performance monitoring task for the `sin` filter
function monitorFilter(code) {
    if (code === 0) {
        // record starting function time
        console.time('monitor');
        return
    }
    // calculates the `sin` duration
    // and prints it to the console
    console.timeEnd('monitor');
}
```

Figura 100 - Código JS com a monitorização de desempenho para a aplicação dos filtros

```
Output (picture1.jpg):
> `monitor: 17254.9228515625 ms`
Output (picture2.jpg):
> `monitor: 16924.600830078125 ms`
```

Figura 101 - Resultado na consola da monitorização de desempenho da aplicação do filtro "sin" nas imagens de teste

6.2.3. Transformação – *Caching*

Para a mesma função foi implementada uma transformação de *caching* que tem como objetivo armazenar o resultado obtido com o cálculo do valor do seno correspondente a uma dada cor (Figura 102). Para isso foi necessário alterar o JS para que estendesse a memória inserida no módulo WASM de forma a ter espaço para armazenar os valores de *cache*. Depois disso, foi só alterar a função que retorna o resultado da cor (`color_sin`) para que devolvesse o valor da *cache* caso já tivesse sido calculado. Neste exemplo, o processo de *caching* está a ser feito sobre uma nova função adicionada, no entanto, poderia ter sido adicionado um *advice* para o implementar num dos filtros já existente no código.

```
(...)
aspects:
  context:
    functions:
      (...)
      color_sin:
        args:
          - name: x
            type: i32
          - name: cachePtr
            type: i32
        result: i32
        variables:
          res: i32
        code: >
          (block $label$0
           (br_if $label$0
            (i32.eqz
             (local.tee %res%
              (i32.load8_u
               (i32.add
                (local.get %cachePtr%)
                (local.get %x%)
               )
              )
             )
            )
           )
           (return (local.get %res%))
          )
          (i32.store8
           (i32.add
            (local.get %cachePtr%)
            (local.get %x%)
           )
```

```
            (local.tee %res%
             (i32.trunc_f64_u
              (f64.min
               (f64.mul
                (f64.abs
                 (call %sin%
                  (f64.convert_i32_u
                   (local.get %x%)
                  )
                 )
                )
                (f64.mul
                 (f64.convert_i32_u
                  (local.get %x%)
                 )
                 (f64.const 2.333333333)
                )
               )
               (f64.const 255)
              )
             )
            )
           )
           (return (local.get %res%))
      sin_modification:
        exported: sin
        args:
          - name: width
            type: i32
          - name: height
            type: i32
          - name: cachePtr
            type: i32
        variables:
          i: i32
        code: >
          (...) ;; implementação deve incluir `cachePtr` na chamada à função `c
olor_sin`
      (...)
(...)
```

Figura 102 - Código da transformação de *caching* para o filtro "sin"

O resultado desta otimização pode ser verificado na Figura 103. Como é possível observar a implementação da *cache* aumentou consideravelmente o desempenho desta função.

```
Output (picture1.jpg):
> `monitor: 385.27490234375 ms`
Output (picture2.jpg):
> `monitor: 4789.6279296875 ms`
```

Figura 103 - Resultado na consola da aplicação do filtro "sin" com *cache* para as duas imagens de teste

6.2.4. Transformação - Reparação de um Erro

A próxima transformação está relacionada com uma atualização realizada à expressão utilizada para calcular o valor da cor do pixel, que por sua vez, passou a incluir um problema que poderia aparecer caso a cor de entrada provocasse uma divisão por zero ($\min(255, abs(\frac{x}{\frac{1}{3000}x}\ sin\ x))$). Com isto, a transformação consiste numa correção temporária que tem como objetivo evitar esta divisão por zero, e como consequência, corrigir o problema que estava a ser despoletado.

Para isso foi criado um *template* que visa encontrar divisões do tipo *float 64-bit*. Utilizando este template foi adicionado um *advice* que protegia estas divisões ao validar se o resultado do denominador seria zero ou não. Caso fosse zero, o valor seria substituído por "1", mantendo assim o resultado igual ao valor do numerador. Com esta condição o código modificado ficou devidamente protegido. O código com a alteração da função e a implementação da transformação encontra-se ilustrado na Figura 104.

```
templates:
  zeroDivF64: "(f64.div %firstOperand% %secondOperand%)"
aspects:
  context:
    functions:
      (...)
      color_sin:
        args:
          - name: x
            type: i32
        result: i32
        variables:
          res: i32
        code: >
          (i32.trunc_f64_s
           (f64.min
            (f64.mul
             (f64.abs
              (call %sin%
               (f64.convert_i32_u
                (local.get %x%)
               )
              )
             )
             (f64.div
              (f64.convert_i32_u
               (local.get %x%)
              )
              (f64.mul
               (f64.convert_i32_u
                (local.get %x%)
               )
               (f64.const 0.0033333)
              )
             )
            )
            (f64.const 255)
           )
```

```
          )
      (...)
  advices:
    (...)
    zeroDivFix:
      all: true
      order: 5
      pointcut: () => template(zeroDivF64)
      variables:
        divValue: f64
      advice: >
        (f64.div
         %zeroDivF64:select(firstOperand)%
         (select
          (f64.const 1)
          (local.tee %divValue% %zeroDivF64:select(secondOperand)%)
          (f64.eq
           (local.get %divValue%)
           (f64.const 0)
          )
         )
        )
```

Figura 104 - Código da transformação que corrige o erro na expressão do filtro "sin"

6.2.5. Transformação - Filtro *Sepia*

A próxima transformação tem como objetivo apresentar uma utilização das funcionalidades do *pointcut* e dos *templates* com maior detalhe. Para isso, foi criado uma função para o filtro *Sepia* e uma outra que executa este filtro cem mil vezes para a mesma imagem (Figura 105). O número de vezes selecionado foi arbitrário, servindo apenas para perceber se este filtro torna a imagem mais clara ou escura. Neste caso, tornou-se mais clara, uma vez que com a repetição do filtro, a imagem ficou com os píxeis pintados a branco.

```
aspects:
  context:
    functions:
      (...)
      sepia_modification:
        exported: sepia
        args:
          - name: width
            type: i32
          - name: height
            type: i32
        variables:
          i: i32
          r: i32
          g: i32
          b: i32
        code:
          (local.set %width%
           (i32.shl
            (i32.mul
             (local.get %width%)
              (local.get %height%))
```

```
            (i32.const 2)))
          (loop $L0
           (if $I1
            (i32.gt_s
             (local.get %width%)
             (local.get %i%))
            (then
            (local.set %r%
             (i32.load8_u
              (local.get %i%)))
            (local.set %g%
             (i32.load8_u offset=1
              (local.get %i%)))
            (local.set %b%
             (i32.load8_u offset=2
              (local.get %i%)))
             (i32.store8
              (local.get %i%)
              (i32.trunc_f32_u
               (f32.min
                (f32.add
                 (f32.add
                  (f32.mul (f32.convert_i32_s (local.get %r%)) (f32.const 0.393
))
                  (f32.mul (f32.convert_i32_s (local.get %g%)) (f32.const 0.769
))
                 )
                 (f32.mul (f32.convert_i32_s (local.get %b%)) (f32.const 0.189)
)
                )
                (f32.const 255)
               )
              ))
             (i32.store8 offset=1
              (local.get %i%)
              (i32.trunc_f32_u
               (f32.min
                (f32.add
                 (f32.add
                  (f32.mul (f32.convert_i32_s (local.get %r%)) (f32.const 0.349
))
                  (f32.mul (f32.convert_i32_s (local.get %g%)) (f32.const 0.686
))
                 )
                 (f32.mul (f32.convert_i32_s (local.get %b%)) (f32.const 0.168)
)
                )
                (f32.const 255)
               )
              ))
             (i32.store8 offset=2
              (local.get %i%)
              (i32.trunc_f32_u
               (f32.min
                (f32.add
                 (f32.add
                  (f32.mul (f32.convert_i32_s (local.get %r%)) (f32.const 0.272
))
```

```
                  (f32.mul (f32.convert_i32_s (local.get %g%)) (f32.const 0.534
))
                 )
                 (f32.mul (f32.convert_i32_s (local.get %b%)) (f32.const 0.131)
)
                )
                (f32.const 255)
               )
              ))
             (local.set %i%
              (i32.add
               (local.get %i%)
               (i32.const 4)))
             (br $L0))))
      sepia_filter_100000_times:
        exported: sepia_100000_times
        args:
          - name: width
            type: i32
          - name: height
            type: i32
        variables:
          i: i32
        code:
          (local.set %i% (i32.const 100000))
          (block $label$0
           (br_if $label$0
            (i32.eqz (local.get %i%))
           )
           (loop $label$1
            (call %sepia_modification%
             (local.get %width%)
             (local.get %height%)
            )
            (br_if $label$1
             (i32.ne
              (local.tee %i%
               (i32.add (local.get %i%) (i32.const -1))
              )
              (i32.const 0)
             )
            )
           )
          )
```

Figura 105 - Código com a implementação das funções "sepia" e "sepia_100000_times"

A transformação baseia-se na aplicação de duas otimizações ao código da função `sepia_100000_times`. A primeira otimização consiste na substituição da instrução com a chamada à função *Sepia* pelo respetivo código contido na função. Com isto pretende-se remover a sobrecarga que normalmente ocorre na chamada a funções (Jangda, Power, Emery, & Guha, 2019). A segunda otimização baseia-se num técnica conhecida por *Loop Unrolling* (Kukunas, 2015), onde muito resumidamente, se tenta minimizar a sobrecarga existente durante a iteração, ao reduzir o número total de iterações do ciclo. Para isso, o corpo da função terá de ser repetido, e a atualização do iterador terá de ter em atenção o

respetivo número de repetições efetuadas. Após ambas terem sido aplicadas, o código da função deve ser capaz de manter a funcionalidade mesmo possuindo um ciclo com menor iterações e sem qualquer chamada à função *Sepia*.

6.2.5.1. Atualização da Monitorização de Desempenho

O primeiro passo para esta transformação consiste na atualização dos *advices* de monitorização para que suportem as novas funções acrescentadas ao programa (Figura 106). Neste passo, foi também atualizada a função de monitorização para suportar um novo parâmetro que indica se a monitorização está a ser feita sobre uma função que filtra a imagem ou sobre a função que executa múltiplas vezes o filtro *Sepia* sobre a mesma imagem. Para isso, foi necessário criar a seguinte condição no primeiro argumento da chamada à função (dentro do *advice*): `%"0":assert((x) => func.Name == "sepia_100000_times")%%"1":assert((x) => func.Name != "sepia_100000_times")%`. Esta condição é formada por duas *static expressions* que utilizam a função `assert` para imprimir o valor de entrada caso a condição seja verdadeira. Neste caso, se a primeira condição for verdadeira, ou seja, se o nome da função, obtido no contexto produzido pelo *pointcut* `func`, for "sepia_100000_times", então a primeira condição imprime "0" e a segunda não imprime nada, caso contrário, a primeira função não imprime nada e a segunda imprime "1". Para além disso, foi alterada a expressão no *pointcut* para incluir as novas funções adicionadas ao módulo.

```
aspects:
  context:
    functions:
      monitoring:
        args:
          - name: fn_type
            type: i32
          - name: type
            type: i32
        imported:
          module: env
          field: monitoring
      (...)
  advices:
    monitor_around_start:
      all: true
      order: 0
      pointcut: () => func(void sepia_100000_times(..)) || func(void sepia(..))
 || func(void sin(..)) || (func(void *(..), exported) && template(t, true))
      advice: >
        (call %monitoring% (i32.const %"0":assert((x) => func.Name == "sepia_10
0000_times")%%"1":assert((x) => func.Name != "sepia_100000_times")%) (i32.const
 0))
        %this%
    monitor_end:
      all: true
      order: 1
      pointcut: () => (func(void sepia_100000_times(..)) || func(void sepia(..)
) || func(void sin(..)) || func(void *(..), exported) && template(t, true)) &&
returns(void)
```

```
      advice: >
        (call %monitoring% (i32.const %"0":assert((x) => func.Name == "sepia_10
0000_times")%%"1":assert((x) => func.Name != "sepia_100000_times")%) (i32.const
 1))
        %this%
```

Figura 106 - Código com a atualização da transformação para a monitorização de desempenho das funções de filtro

O código implementado para a função de monitorização encontra-se ilustrado na Figura 107. Relativamente à execução da ferramenta, devido ao elevado número de iterações, foi reduzido o tamanho da imagem em cem vezes (*43.2x76.8* píxeis). O tempo de execução desta função pode ser visualizado na Figura 108.

```
// handles the performance monitoring task for the `sin` filter
function monitorFilter(fn_code, code) {
    if (code === 0) {
        // record starting function time
        console.time(fn_code === 0 ? 'monitor_all' : 'monitor');
        return
    }
    // calculates the `sin` duration
    // and prints it to the console
    console.timeEnd(fn_code === 0 ? 'monitor_all' : 'monitor');
}
```

Figura 107 - Código JS com a nova função de monitorização de desempenho das funções de filtro

```
Output:
> `monitor_all: 6309.0439453125 ms`
```

Figura 108 - Resultado na consola após execução da função que aplica cem mil vezes o filtro *sepia*

6.2.5.2. Remoção das Chamadas

O próximo passo consistiu na substituição da chamada à função *Sepia* pelo próprio código (Figura 109). Esta transformação foi feita com o intuito de tentar otimizar o desempenho da função ao reduzir a sobrecarga que uma chamada acarreta. Para este efeito foi necessário a criação de dois *advices*, em que o primeiro é responsável por armazenar o valor inicial da variável usada para iterar no ciclo, e o segundo serve para substituir a chamada pelas instruções da função.

```
aspects:
  advices:
    (...)
    improve_sepia_modification_100000_times_start:
      all: true
      order: 3
      pointcut: (i32.param[0] W) => func(void sepia_100000_times(..))
      variables:
        w: i32
      advice: >
```

```
        (local.set %w% (local.get %W%))
        %this%
    improve_sepia_modification_100000_times_remove_call:
      all: true
      order: 4
      pointcut: (i32.param[0] W, i32.local[w] w) => call(void sepia(..))
      variables:
        j: i32
        r: i32
        g: i32
        b: i32
      advice: >
        %call.callee.code:map((x) => "(local.set %W% (local.get %w%))";"(local.
set \%i\% (i32.const 0))";x):string():replace("\%i\%", "\%j\%"):remove("(return
)")%
```

Figura 109 - Código da transformação com a substituição das chamadas pelo conteúdo da função "sepia"

Para o primeiro *advice* foi utilizada a função `func` na expressão do *pointcut*. A configuração definida na função serviu para que fosse obtida apenas a função `sepia_100000_times`. De seguida, foi criada uma variável local que serviu para armazenar o valor inicial do iterador. Neste caso, este valor é proveniente do primeiro parâmetro da função. Assim, foi necessário passar o valor deste parâmetro para o *advice* através dos argumentos passados ao contexto do *pointcut*. Por fim, este *advice* simula o comportamento de um "*before*", isto é, a instrução de atribuição do valor surgirá antes das instruções presentes no código da função.

Relativamente ao segundo *advice*, este recorreu à função `call` na expressão do *pointcut* uma vez que esta disponibiliza o código da função que se encontra a ser chamada. Com isto, o *advice* substitui as instruções com a chamada à função *Sepia* pelo código presente nessa função. Para que isto resultasse foi necessário realizar duas transformações. A primeira consiste no espelhamento das variáveis utilizadas na função Sepia. Neste caso, foram criadas as variáveis `r`, `g` e `b` que espelham de forma direta essas variáveis, no entanto, através da função `replace`, foi feita a substituição da variável `i` pela variável `j` (também criada no *advice*), uma vez que esta variável já estava a ser utilizada na função `sepia_100000_times`. Por fim, é necessário adicionar a instrução que preenche o iterador com o respetivo valor inicial, antes da execução do código adicionado uma vez que por cada vez que o filtro é aplicado, o iterador para os píxeis tem de ser reiniciado.

O tempo de execução obtido após a execução da função `sepia_100000_times` sem as chamadas à função *Sepia* encontra-se ilustrado na Figura 110. Apesar do tempo obtido ter sido melhor que o anterior à transformação, isto não significa que tenha sido uma melhoria válida para o programa, uma vez que a afirmação só é verdadeira se for devidamente comprovada, da realização de uma maior quantidade e variedade de testes. No entanto, para o contexto deste exemplo, vai-se considerar que esta alteração realmente foi uma melhoria no desempenho do programa.

```
Output:
> `monitor_all: 6290.072021484375 ms`
```

Figura 110 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* com a remoção das chamadas ao "sepia" implementado

6.2.5.3. Loop Unrolling

Por último, foi aplicada a técnica de otimização *Loop Unrolling*, onde foi necessário recorrer aos *templates* disponibilizados pela ferramenta para que fosse possível repetir múltiplas vezes o "corpo" do ciclo, e ainda atualizar a variável que controla o número de iterações deste ciclo.

Foi criado um conjunto de *templates* (Figura 111) que têm como objetivo não só refletir os vários passos existentes no ciclo da função `sepia_100000_times`, como também qualquer função que possua uma estrutura equivalente. O *template* `loop_t` é o ponto de partida para a estrutura de *templates* criada, possuindo assim a base estrutural do ciclo. Este começa por inicializar o índice `index`. De seguida, é definido o identificador `after_set` que contém todas as instruções que procedem à inicialização do índice e antecedem o elemento `block` que está associado ao ciclo da função. Este é um identificador opcional, ou seja, tanto pode existir valores que coincidam com este como não. Isto acontece porque antes do identificador não existe qualquer caractere separador, isto é, espaços ou *tabs*. Depois disso, é definido o elemento `block` do ciclo. Este é composto por uma *label* (identificador `block_label`), por uma condição inicial (identificador `initial_condition`), pelo respetivo ciclo e por um elemento opcional que visa coincidir com todas as instruções que se sucedam ao ciclo (identificador `after_loop`). O ciclo em si é também composto por uma *label* (identificador `label`), pelo corpo do ciclo (identificador `body`) e pela condição de paragem. Para evitar erros no código, foi criada uma restrição para o corpo do ciclo. Esta restrição foi feita com recurso à função `not_include_one`, onde foram incluídos dois *templates* (`changeIndex1` e `changeIndex2`), que ditam que no corpo do ciclo não deve existir qualquer alteração ao índice utilizado. Como se pode verificar, ambos os *templates* devem possuir o mesmo identificador declarado no *template* base (`index`) para que a restrição aconteça. Na condição do ciclo é utilizado novamente o identificador `label`, exigindo assim que tanto a *label* do ciclo como a *label* presente na condição sejam iguais. A condição inclui também a instrução que decrementa o índice da função. Mais uma vez, foi utilizado um identificador que já tinha sido definido no *template*, o `index`, referindo-se assim ao índice do ciclo. Por fim, a instrução que serve de atualização ao índice é validada com recurso ao método `includes_one`, que importa dois *templates* diferentes, que têm como objetivo o decremento do índice por uma unidade. Para que fosse possível obter o valor do decremento, foi necessário recorrer ao método `defines`, que ao estar a seguir ao método `includes_one`, estabelece que ambos os templates devem definir o identificador em questão. Desta forma, o identificador `incrementor` deve ser definido em ambos os *templates*, sendo que neste caso, vai assumir o valor do decremento do índice ("1" ou "-1").

```
templates:
  loop_t: >
    (local.set %index% (i32.const 100000))%after_set%
    (block %block_label%
     %initial_condition%
     (loop %label%
      %body:not_includes_one(changeIndex1, changeIndex2)%
      (br_if %label%
       (i32.ne
        (local.tee %index% %next_index:includes_one(nextIndex1, nextIndex2):def
ines(incrementor)%)
        (i32.const 0)
       )
      )
     )%after_loop%
    )
  changeIndex1: "(local.set %index% [_])"
  changeIndex2: "(local.tee %index% [_])"
  nextIndex1: "(i32.sub (local.get %index%) (i32.const %incrementor:includes(in
crementor_pos)%))"
  nextIndex2: "(i32.add (local.get %index%) (i32.const %incrementor:includes(in
crementor_neg)%))"
  incrementor_pos: "1"
  incrementor_neg: "\\-1"
aspects:
    (...)
    improve_sepia_modification_100000_times_repeat:
      all: true
      order: 5
      pointcut: () => template(loop_sepia_100000)
      advice: >
        (local.set %loop_sepia_100000:select(index)% (i32.const 100000))
        %loop_sepia_100000:select(after_set)%
        (block %loop_sepia_100000:select(block_label):map((l) => "$out";l)%
         %loop_sepia_100000:select(initial_condition):replace("%loop_sepia_1000
00:select(block_label)%", "%loop_sepia_100000:select(block_label):map((l) => "$
out";l)%")%
         (loop %loop_sepia_100000:select(label):map((l) => "$out";l)%
          %loop_sepia_100000:select(body):repeat(100)%
          (br_if %loop_sepia_100000:select(label):map((l) => "$out";l)%
           (i32.ne
            (local.tee %loop_sepia_100000:select(index)% %loop_sepia_100000:sel
ect(next_index):replace(incrementor, "%loop_sepia_100000:select(incrementor):ma
p((x) => x * 100)%")%)
            (i32.const 0)
           )
          )
         )%loop_sepia_100000:select(after_loop)%
        )
```

Figura 111 - Código da transformação que aplica a otimização *Loop Unrolling* à função que aplica cem mil vezes o filtro *sepia*

Com isto, o *advice* criado na Figura 111 utiliza estes *templates* para reconstruir a função de forma a que seja possível reduzir o número de iterações necessárias para o ciclo. Nesta reconstrução recorre-se frequentemente à função `select` para aceder aos identificadores

definidos no *template*. É adicionado o prefixo `$out` aos valores das *labels* para que sejam evitados conflitos com as *labels* já existentes na função. O corpo do ciclo é repetido cem vezes através da função `repeat`. Por fim, o valor que decrementa o índice é multiplicado por cem.

O resultado da execução da função com as transformações aplicadas encontra-se ilustrada na Figura 112, sendo que também foi executada uma versão com o ciclo repetido apenas dez vezes ao invés de cem. Mais uma vez, apesar do tempo ser menor que o tempo registado acima, pode não significar que seja uma implementação mais eficiente.

```
Output (x100):
> `monitor_all: 6277.572998046875 ms`
Output (x10):
> `monitor_all: 6286.965087890625 ms`
```

Figura 112 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* após a otimização *Loop Unrolling* realizada para a mesma

Foi também implementada a otimização *Loop Unrolling* para a função `sepia` invés da função `sepia_100000_times`. Para isso foram adicionados um conjunto de *templates* com uma estrutura semelhante aos anteriores, no entanto, muito mais específicos à função que irá sofrer a alteração. Esta alteração encontra-se ilustrada na Figura 113.

```
templates:
  (...)
  loop_sepia: >
    (local.set %total% %totalAssign%)
    (loop %label%
     (if
      (i32.gt_s
       (local.get %total%)
       (local.get %index%))
      (then
       %body%
       (local.set %index%
        (i32.add
         (local.get %index%)
         (i32.const %incrementor%)))
        (br %label%)
      )
     )
    )
aspects:
  (...)
  advices:
    (...)
    improve_sepia_modification:
      all: true
      order: 2
      pointcut: () => func(void sepia(..)) && template(loop_sepia)
      advice: >
        (local.set %loop_sepia:select(total)% %loop_sepia:select(totalAssign)%)
        (loop %loop_sepia:select(label)%
```

```
         (if
          (i32.gt_s
           (local.get %loop_sepia:select(total)%)
           (local.get %loop_sepia:select(index)%))
          (then
           %loop_sepia:select(body):repeat(100):map((b, index) => b:replace("(l
ocal.get \%i\%)", "(i32.add (local.get \%i\%) (i32.const %index:map((i) => i *
"%loop_sepia:select(incrementor)%")%))")")%
           (local.set %loop_sepia:select(index)%
            (i32.add
             (local.get %loop_sepia:select(index)%)
             (i32.const %loop_sepia:select(incrementor):map((x) => x * 100)%)))
           (br %loop_sepia:select(label)%)
          )
         )
        )
        (loop %loop_sepia:select(label):map((l) => l; "1")%
         (if
          (i32.gt_s
           (local.get %loop_sepia:select(total)%)
           (local.get %loop_sepia:select(index)%))
          (then
           %loop_sepia:select(body)%
           (local.set %loop_sepia:select(index)%
            (i32.add
             (local.get %loop_sepia:select(index)%)
             (i32.const %loop_sepia:select(incrementor)%)))
           (br %loop_sepia:select(label):map((l) => l; "1")%)
          )
         )
        )
    (...)
```

Figura 113 - Código da transformação que aplica a otimização *Loop Unrolling* à função *sepia*

Depois de várias iterações, percebeu-se que a melhor configuração para a transformação seria a combinação de todas as otimizações, com dez repetições para ambas as funções. O resultado da consola encontra-se ilustrado na Figura 114.

```
Output (x10):
> `monitor_all: 6186.10595703125 ms`
```

Figura 114 - Resultado na consola da execução da função que aplica cem mil vezes o filtro *sepia* após a otimização *Loop Unrolling* realizada na função *sepia*

6.2.6. Transformação – Melhoramentos com JS

Atualmente, é quase impossível alcançar o desempenho necessário para executar aplicações baseadas em cálculos intensivos através do JS. Estas aplicações podem ser jogos, edição de vídeo, renderização 3D ou produção musical. Desta forma, trazer tais aplicações para a *web* é um problema, e é por essa razão que a utilização de JS em conjunto com WASM é uma opção muito adequada para a *web*. JS como se sabe é a linguagem de programação interpretada pelos navegadores que permite a criação de páginas *web* interativas, e WASM é uma linguagem de programação criada para ser rápida, portátil e flexível. A combinação

destas linguagens permitirá a implementação deste tipo de aplicações na *web*, alcançando assim marcos que antes não eram possíveis (Mihajlija, 2019).

WASM e JS são interoperáveis, partilhando assim funções, memórias, variáveis e objetos. Sabendo isto, fez todo o sentido incluir extensões às funcionalidades disponibilizadas pelo WASM recorrendo ao JS (ICHI.PRO, 2021). Assim, foi adicionada uma nova camada JS que faz a ligação entre os clientes JS e o módulo WASM. Esta camada não só é responsável por fazer a gestão das novas funcionalidades implementadas no WASM, tais como, novos tipo de dados, execução de *runtime expressions*, e gestão da memória associada a estas transformações, como também permite que as funções importadas e exportadas pelo cliente suportem estes novos tipos definidos.

Sabendo isto, recorreu-se às extensões *runtime* disponibilizadas pela ferramenta para permitir que o registo e monitorização das funções possa ser feito de forma concorrente, e que possam existir vários mecanismos de *cache* para qualquer função desejada.

6.2.6.1. *Logging* e Monitorização Partilhado

Para que o registo e monitorização dos filtros aplicados possa ser feito de forma concorrente é necessário que a função JS invocada para o efeito tenha conhecimento do tipo de filtro que a está a invocar.

O processo de identificação do tipo da função poderia ser feito através da utilização de códigos numéricos, no entanto, esta implementação englobava a manutenção e sincronização dos tipos para ambas as aplicações, ou seja, os códigos atribuídos a cada função do módulo WASM precisavam de ser rotulados no lado do cliente para que os conseguisse identificar devidamente. Para evitar isto, utilizou-se o nome da função presente no contexto disponibilizado pelo *pointcut*, que é passado como um argumento do tipo *string*. Assim, todas as funções são automaticamente rotuladas e identificadas na função JS. A transformação pode ser visualizada na Figura 115.

```
templates:
  t: "(i32.store%_%)"
aspects:
  context:
    functions:
      monitoring:
        args:
          - name: fn_name
            type: string
          - name: state
            type: string
        imported:
          module: env
          field: monitoring
      (...)
  advices:
    monitor_around_start:
      all: true
      order: 0
```

```
      pointcut: () => func(void sepia(..)) || func(void sin(..)) || func(void *
(..), exported) && template(t, true)
      advice: >
        (call %monitoring% /"%func.Name%"/ /"start"/)
        %this%
    monitor_end:
      all: true
      order: 1
      pointcut: () => (func(void sepia(..)) || func(void sin(..)) || func(void
*(..), exported) && template(t, true)) && returns(void)
      advice: >
        (call %monitoring% /"%func.Name%"/ /"end"/)
        %this%
```

Figura 115 - Código da transformação que permite a monitorização partilhado pela mesma função

Esta alteração forçou o cliente JS a substituir a utilização direta do módulo WASM pela camada intermédia JS. O código com esta alteração, e adicionalmente, a alteração da função de registo e monitorização, está ilustrado na Figura 116.

```
const importObject = {
    env: {
        (...)
        monitoring: (fn_code, code) => monitorFilter(fn_code, code),
        (...)
    }
};

const wmr = await import('./output.js');
const { instance, model } = (await wmr.loadWasm("./output.wasm", importObject))
;
const wasm = instance.exports;

//...

// handles the performance monitoring task for a filter
function monitorFilter(fnName, state) {
    switch (state) {
        case 'start':
            // record starting function time
            console.time(fnName);
            break;
        case 'end':
            // calculates the function duration
            // and prints it to the console
            console.timeEnd(fnName);
            break;
        default:
            // logs unknown state
            console.log(`Unknown state "${state}" on function "${fnName}"`);

            // close any open timer for the function
            console.timeEnd(fnName);
    }
}
```

Figura 116 - Código JS com as alterações para implementar a monitorização partilhado

6.2.6.2. *Cache* Múltipla

Outra transformação que precisou de ser implementada com o recurso às extensões *runtime* foi a implementação de um mecanismo de *cache*, especificado apenas uma vez, que servirá para aplicar num conjunto de funções identificadas pelo utilizador.

Para isso foi criado um mapa onde a chave consiste no nome da função, e o valor consiste na posição inicial de memória da *cache* associada à respetiva função. Depois foram criados três *advices* que através de *templates* permitem implementar o mecanismo de *cache* no código. O primeiro procura as instruções de armazenamento que não possuem o *offset* definido e adiciona esse *offset* com o valor zero. O segundo inicializa a *cache* na função caso ainda não exista registada. E por fim, o terceiro faz todo o processo de consulta e armazenamento dos valores na *cache* utilizando os valores do *offset* (incluindo os valores anteriormente adicionados). O código da transformação encontra-se ilustrado na Figura 117.

```
templates:
  (...)
  store: "(i32.store8 %offset:includes_all(offset):defines(offsetValue)% %index
% %value%)"
  storeNoOffset: "(i32.store8 %index:not_includes(offset)% %value%)"
  offset: "offset=%offsetValue%"
  condition: >
    (i32.gt_s
     (local.get %_%)
     (local.get %loop_index%))
aspects:
  context:
    variables:
      cachePtr: map[string]i32
      cachePtrAccum: i32
  advices:
    (...)
    addOffset:
      all: true
      order: 2
      pointcut: () => template(storeNoOffset)
      variables:
        res: i32
      advice: >
        (i32.store8 offset=0 %storeNoOffset:select(index)% %storeNoOffset:selec
t(value)%)
    save_cache_ptr:
      all: true
      order: 3
      pointcut: (i32.param[0] w, i32.param[1] h) => (func(* sin(..)) || func(*
invert(..))) && template(condition, true) && template(store, true)
      variables:
        last: i32
        cachePtrVal: i32
      advice: >
        (local.set %last%
         (i32.shl
          (i32.mul (local.get %w%) (local.get %h%))
```

```
          (i32.const 2)
         )
        )
        (block $B0
         (br_if $B0
          (i32.ne /!!#cachePtr["%func.Name%"]/ (i32.const 0))
         )
         (global.set #cachePtr["%func.Name%"] /#last+#cachePtrAccum*256*3/)
         (global.set #cachePtrAccum /#cachePtrAccum+1/)
        )
        (local.set #cachePtrVal /#cachePtr["%func.Name%"]/)
        %this%
    apply_individual_cache:
      all: true
      order: 4
      pointcut: (i32.local[cachePtrVal] cachePtrVal) => (func(* sin(..)) || fun
c(* invert(..))) && template(condition, true) && template(store)
      variables:
        res: i32
        color: i32
        cacheIndex: i32
      advice: >
        (local.set %color%
         (i32.load8_u
          (i32.add
           (local.get %condition:select(loop_index)%)
           (i32.const %store:select(offsetValue)%)
          )
         )
        )
        (if
         (i32.eqz
          (local.tee %res%
           (i32.load8_u
            (local.tee %cacheIndex%
             (i32.add
              (i32.add
               (local.get %cachePtrVal%)
               (i32.mul
                (i32.const 256)
                (i32.const %store:select(offsetValue)%)
               )
              )
              (local.get %color%)
             )
            )
           )
          )
         )
         (then
          (i32.store8
           (local.get %cacheIndex%)
           (local.tee %res% %store:select(value)%)
          )
         )
        )
        (i32.store8 %store:select(offset)% %store:select(index)% (local.get %re
s%))
```

Figura 117 - Código da transformação para inclusão de uma *cache* para múltiplas funções

6.3. Markdown *Parser*

O próximo exemplo engloba a modificação de um aplicação que faz o *parse* de texto para o formato Markdown, renderizando o resultado na forma de elementos HTML. A modificação descrita neste capítulo consiste numa extensão ao programa, que permite que seja suportada a contagem dos diferentes tipos analisados. O código da aplicação encontra-se publicado no Github (Andersson, 2021).

Numa fase inicial, para que fosse possível realizar esta transformação foi necessário proceder à análise do código. Para isso foi adicionada uma função que tem como objetivo imprimir na consola uma mensagem do tipo *string*. Neste caso, foi definido um *advice* que insere uma chamada a esta função no início de todas as funções existentes no código. A mensagem passada como argumento consiste no nome da função e o valor de cada parâmetro recebido (por exemplo, `$f1 0 7`, onde `$f1` é o nome da função, e `0` e `7` são os valores dos parâmetros recebidos pela função). Com isto, foi possível perceber parte do fluxo do programa necessário à modificação. O código da transformação está ilustrado na Figura 118, e a respetiva análise encontra-se representada na Tabela 8, contendo para cada tipo de elemento o exemplo utilizado, assim como parte do resultado da consola, e a regra utilizada para realizar a sua contagem. Parte dos exemplos utilizados nesta tabela foram retirados da documentação da ferramenta *markdown-wasm*.

```
aspects:
  context:
    functions:
      debug:
        args:
          - name: message
            type: string
        imported:
          module: env
          field: debug
  advices:
    debug:
      pointcut: () => func(* * (..))
      advice: >
        (call %debug% /"%func.name%"%func.params:map((v) => "+ ' ' +"; "#"; v)%
/)
        %this%
```

Figura 118 - Código da transformação para análise do programa *markdown-wasm*

Tabela 8 - Resultados da análise realizada ao fluxo do programa *markdown-wasm*

Elemento	***Exemplo***	***Resultado***	***Regra***
Títulos	No Header # This is H1 ## This is H2 ### This is H3 #### This is H4 ##### This is H5 ###### This is H6	> cat output | grep f83 -A1 | grep f1 | sort --unique *f1 5250872 1559* *f1 5250872 1712* *f1 5250872 1707* *f1 5250872 1702* *f1 5250872 1697* *f1 5250872 1692* *f1 5250872 1687*	**Fluxo:** *f83 → f81* 2º argumento: • 1559 – Parágrafo. • 1712 – Título 1. • 1707 – Título 2. • 1702 – Título 3. • 1697 – Título 4. • 1692 – Título 5. • 1687 – Título 6.
Bold e Itálico	**strong** or __strong__ (Cmd + B) *emphasize* or _emphasize_ (Cmd + I)	> cat output | grep f85 | sort --unique f85 0 0 5250872 f85 1 0 5250872	Fluxo: *f85* 1º argumento: • 0 – *Bold*. • 1 – Itálico.
Listas	* *Unordered list item* 1. Ordered list item - [ ] Task 1 121) Ordered lists can start	> cat output | grep f83 -A1 | grep f1 | sort --unique *f1 5250872 1437* *f1 5250872 1628* *f1 5250872 2201* *f1 5250872 2344* *f1 5250872 2357*	**Fluxo:** Para ter em conta os sub-ítems, a regra foi alterada para *f6 && arg[2] != 5 && arg[2] != 6 → f83 → f81* 1º argumento: • 1437 – Item da lista do tipo *checkbox*. • 1628 – Item da lista do tipo texto. • 2201 – Lista ordenada

Elemento	*Exemplo*	*Resultado*	*Regra*
			terminada em “)”. • 2344 – Lista não ordenada. • 2357: Lista ordenada terminada em “.”.
Rasurado	~Strikethrough 1~ ~Strikethrough 2~	> cat output | grep f85 | sort --unique *f85 5 0 5250872*	Fluxo: *f85* 1º argumento: • 5 – Rasurado.
Links	An email example@example.com link. Simple inline link <http://chenluois.com>, another inline link [Smaller](http://smallerapp.com), one moreinline link with title [Resize](http://resizesafari.com "a Safari extension"). A [reference style][id] link. Input id, then anywhere in the doc, define the link with corresponding id: [id]: http://mouapp.com "Markdown editor on Mac OS X"	> cat output | grep f85 | sort --unique *f85 2 5250000 5250872*	Fluxo: *f85* 1º argumento: • 2 – *Link*.
Citações em Bloco	> Right angle brackets > are used for block quotes.	> cat output | grep f83 -A1 | grep f1 | sort --unique *f1 5250872 1559* *f1 5250872 2384*	**Fluxo:** *f83 → f81* 2º argumento: • 1559 – Parágrafo. • 2384 – Bloco com citação.
Tabelas	A simple table looks like this: First Header | Second Header | Third Header ----- | ----- | -----	> cat output | grep f83 -A1 -B1 | grep f13 -A2 | tail -n 1 | sort --unique	**Fluxo:** *f83 → f81* 2º argumento: • 2428– Tabela.

Elemento	*Exemplo*	*Resultado*	*Regra*
	Cell | Cell | Cell Cell | Cell | Cell If you wish, you can add a leading and tailing pipe to each line of the table: | First Header | Second Header | Third Header | | ----- | ----- | ----- | | Cell | Cell | Cell | | Cell | Cell | Cell | Specify alignement for each column by adding colons to separator lines: First Header | Second Header | Third Header :--------- | :---------: | ---------: Left | Center | Right Left | Center | Right	*f1 5250872 2428*	
Linha Horizontal	*** --- - - - -	> cat output | grep f83 -A1 -B1 | grep f13 -A2 | tail -n 1 | sort --unique *f1 5250872 2326*	**Fluxo:** *f83 → f81* 2º argumento: • 2326 – Linha horizontal.
Blocos de Código	```js function codeBlocks() { return "Can be inserted" } ```	> cat output | grep f83 -A1 | grep f1 | sort --unique *f1 5250872 1402*	**Fluxo:** *f83 → f81* 2º argumento: • 1402 – Bloco de código.
Imagens	An inline image ![Smaller icon](http://smallerapp.com/favicon.ico "Title here"), title is optional. A ![Resize icon][2] reference style image.	> cat output | grep f85 | sort --unique *f85 3 5250000 5250872*	Fluxo: *f85* 1º argumento: • 3 – Imagem.

Elemento	*Exemplo*	*Resultado*	*Regra*
	[2]: http://resizesafari.com/favicon.ico "Title"		
Código Inline	Run shell command `ls` to get ``contents`` of the current `directory`.	> cat output | grep f85 | sort --unique *f85 4 0 5250872*	Fluxo: *f85* 1º argumento: • 4 – Código *inline*.

Com base nestas regras foi possível implementar o código ilustrado na Figura 119 para a transformação com a contagem.

```
aspects:
  start: "(global.set %pathPtr% (memory.grow (i32.const 2)))"
  context:
    variables:
      f6ArgVal: i32
      countArray: "[]i32"
      headerCodes: "map[i32]i32 = [[1712, 6], [1707, 7], [1702, 8], [1697, 9],
[1692, 10], [1687, 11]]"
      listCodes: "map[i32]i32 = [[2201, 16], [2344, 17], [2357, 18]]"
      italicCode: i32 = 0
      boldCode: i32 = 1
      linkCode: i32 = 2
      imageCode: i32 = 3
      codeCode: i32 = 4
      strikethroughCode: i32 = 5
      tableCode: "[]i32 = [2428, 12]"
      hrCode: "[]i32 = [2326, 13]"
      codeBlockCode: "[]i32 = [1402, 14]"
      blockQuoteCode: "[]i32 = [2384, 15]"
      pathPtr: i32 = 0
      pathState: i32 = 0
    functions:
      reset:
        exported: reset
        code: "(global.set #countArray /[%'0':repeat(19):join(',')%]/)"
      get_italic:
        exported: get_italic
        result: i32
        code: "/#countArray[#italicCode]/"
      get_bold:
        exported: get_bold
        result: i32
        code: "/#countArray[#boldCode]/"
      get_link:
        exported: get_link
        result: i32
        code: "/#countArray[#linkCode]/"
      get_image:
        exported: get_image
        result: i32
        code: "/#countArray[#imageCode]/"
      get_inline_code:
        exported: get_inline_code
```

```
        result: i32
        code: "/#countArray[#codeCode]/"
      get_strikethrough:
        exported: get_strikethrough
        result: i32
        code: "/#countArray[#strikethroughCode]/"
      get_headers:
        exported: get_headers
        variables:
          res: "[]i32"
        result: "[]i32"
        code: >
          (local.set #res /Object.entries(#headerCodes).map(([_, i]) => #countA
rray[i] || 0)/)
          (return /[...#res, #res.reduce((accum, count) => accum + count, 0)]/)
      get_tables:
        exported: get_tables
        result: i32
        code: "/#countArray[#tableCode[1]]/"
      get_hrs:
        exported: get_hrs
        result: i32
        code: "/#countArray[#hrCode[1]]/"
      get_code_blocks:
        exported: get_code_blocks
        result: i32
        code: "/#countArray[#codeBlockCode[1]]/"
      get_blocks_quotes:
        exported: get_blocks_quotes
        result: i32
        code: "/#countArray[#blockQuoteCode[1]]/"
      get_lists:
        exported: get_lists
        variables:
          res: "[]i32"
        result: "[]i32"
        code: >
          (local.set #res /Object.entries(#listCodes).map(([_, i]) => #countArr
ay[i] || 0)/)
          (return /[...#res, #res.reduce((accum, count) => accum + count, 0)]/)
      get_count:
        exported: get_count
        result: "[]i32"
        code: "/#countArray/"
  advices:
    fromFnBefore:
      pointcut: () => func(* *(..))
      order: 0
      advice: >
        (global.set %pathState%
         (i32.xor
          (global.get %pathState%)
          (i32.const 1)
         )
        )
        (i32.store8
         (i32.add
          (global.get %pathPtr%)
          (global.get %pathState%)
```

```
          )
          (i32.const %func.order%)
        )
        %this%
    getF6ArgBefore:
      pointcut: (i32.param[2] argVal) => func(* $f6(i32, i32, i32))
      order: 1
      advice: >
        (global.set %f6ArgVal% (local.get %argVal%))
        %this%
    countInline:
      pointcut: (i32.param[0] type) => func(* $f85(i32, i32, i32))
      advice: >
        (global.set #countArray[type] /(#countArray[#type] || 0) + 1/)
        %this%
    countBlocks:
      pointcut: (i32.param[1] argVal) => func(* $f1(i32, i32))
      variables:
        indexVal: i32
      advice: >
        (block $B_CountAll
         (block $B_CountHeaders
          (br_if $B_CountAll
           (i32.ne
            (i32.load8_u
             (i32.add
              (global.get %pathPtr%)
              (global.get %pathState%)
             )
            )
            (i32.const %"$f83":order()%)
           )
          )
          (br_if $B_CountHeaders
           (i32.eqz (local.tee #indexVal /#headerCodes[#argVal]/)))
          (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
          (br $B_CountAll)
         )
         (block $B_CountCodeBlock
          (br_if $B_CountCodeBlock
           (i32.eqz /#codeBlockCode[0]===#argVal/))
          (local.set #indexVal /#codeBlockCode[1]/)
          (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
          (br $B_CountAll)
         )
         (block $B_CountBlockQuote
          (br_if $B_CountBlockQuote
           (i32.eqz /#blockQuoteCode[0]===#argVal/))
          (local.set #indexVal /#blockQuoteCode[1]/)
          (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
          (br $B_CountAll)
         )
         (block $B_CountLists
          (br_if $B_CountLists
           (i32.and
            (i32.eq
```

```
              (i32.load8_u
               (i32.add
                (global.get %pathPtr%)
                (i32.xor (global.get %pathState%) (i32.const 1))
               )
              )
              (i32.const %"$f6":order()%)
             )
             (i32.and
              (i32.ne
               (global.get %f6ArgVal%)
               (i32.const 5)
              )
              (i32.ne
               (global.get %f6ArgVal%)
               (i32.const 6)
              )
             )
            )
           )
           (br_if $B_CountLists
            (i32.eqz (local.tee #indexVal /#listCodes[#argVal]/)))
           (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
           (br $B_CountAll)
          )
          (block $B_CountTables
           (br_if $B_CountAll
            (i32.ne
             (i32.load8_u
              (i32.add
               (global.get %pathPtr%)
               (i32.xor (global.get %pathState%) (i32.const 1))
              )
             )
             (i32.const %"$f13":order()%)
            )
           )
           (br_if $B_CountTables
            (i32.eqz /#tableCode[0]===#argVal/))
           (local.set #indexVal /#tableCode[1]/)
           (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
           (br $B_CountAll)
          )
          (block $B_CountHR
           (br_if $B_CountHR
            (i32.eqz /#hrCode[0]===#argVal/))
           (local.set #indexVal /#hrCode[1]/)
           (global.set #countArray[indexVal] /(#countArray[#indexVal] || 0) + 1/
)
           (br $B_CountAll)
          )
         )
         %this%
```

Figura 119 - Código da transformação com a contagem de elementos no programa *markdown-wasm*

Na Tabela 9 encontra-se detalhada a função de cada um dos elementos expressos na transformação.

Tabela 9 –Elementos presentes na transformação do programa *markdown-wasm*

Tipo	***Elemento***	***Função***
Função inicial	start	Inicializar o espaço de memória que servirá para armazenar as duas últimas funções executadas no programa, antes da atual.
Variáveis Globais	f6ArgVal	Armazena o terceiro argumento presente na última chamada à função "$f6".
	countArray	Array que armazena a contagem dos elementos.
	headerCodes	Mapa utilizado para mapear o código dos "cabeçalhos" para o respetivo índice no *array countArray.*
	listCodes	Mapa utilizado para mapear o código das diferentes "listas" para o respetivo índice no *array countArray*.
	italicCode	Representa o código do elemento "italic" no programa, e também o seu respetivo índice no *array countArray*.
	boldCode	Representa o código do elemento "bold" no programa, e também o seu respetivo índice no *array countArray*.
	linkCode	Representa o código do elemento "link" no programa, e também o seu respetivo índice no *array countArray*.
	imageCode	Representa o código do elemento "imagem" no programa, e também o seu respetivo índice no *array countArray*.
	codeCode	Representa o código do elemento "código inline" no programa, e também o seu respetivo índice no *array countArray*.
	strikethroughCode	Representa o código do elemento "strikethrough" no programa, e também o seu respetivo índice no *array countArray*.
	tableCode	*Array* cujo primeiro elemento é o código da "tabela" no programa, e segundo elemento é o índice da sua contagem no *array countArray*.
	hrCode	*Array* cujo primeiro elemento é o código da "linha horizontal" no programa, e segundo elemento é o índice da sua contagem no *array countArray*.
	codeBlockCode	*Array* cujo primeiro elemento é o código do "bloco de código" no programa, e segundo elemento é o índice da sua contagem no *array countArray*.
	blockQuoteCode	*Array* cujo primeiro elemento é o código do "bloco de citação" no programa, e segundo elemento é o índice da sua contagem no *array countArray*.
	pathPtr	Representa o índice inicial da memória linear que rastreia as duas últimas chamadas a funções.

Tipo	*Elemento*	*Função*
	pathState	Representa o índice atual da memória linear que rastreia as duas últimas chamadas a funções.
Funções	reset	Limpa e prepara o *countArray* para uma nova contagem.
	get_italic	Devolve a contagem para o elemento "italic".
	get_bold	Devolve a contagem para o elemento "bold".
	get_link	Devolve a contagem para o elemento "link".
	get_image	Devolve a contagem para o elemento "imagem".
	get_inline_code	Devolve a contagem para o elemento "código inline".
	get_strikethrough	Devolve a contagem para o elemento "strikethrough".
	get_headers	Devolve um array com a contagem dos vários elementos do tipo "cabeçalho".
	get_tables	Devolve a contagem para o elemento "tabela".
	get_hrs	Devolve a contagem para o elemento "linha horizontal".
	get_code_blocks	Devolve a contagem para o elemento "blocos de código".
	get_blocks_quotes	Devolve a contagem para o elemento "blocos de citação".
	get_lists	Devolve a contagem para o elemento "lista".
	get_count	Devolve um array com a contagem de todos os elementos.
Advices	fromFnBefore	Para cada função, regista na memória linear que a função foi chamada.
	getF6ArgBefore	Obtém o valor do terceiro parâmetro da função "$f6".
	countInline	Faz a contagem dos elementos *inline* através da função "$f85".
	countBlocks	Faz a contagem dos restantes elementos através da função "$f1".

Como já foi mencionado acima, o código encontra-se publicado na *web*, sendo que o código JS que teve de ser alterado para suportar as transformações encontra-se ilustrado na Figura 120. Como é possível verificar, foi alterada a forma como é importado o módulo WASM, passando a ser importada a camada intermédia JS gerada pela ferramenta, a função que é chamada a cada atualização para que cada vez que faz o *parse* do conteúdo, também faça a atualização da contagem, e o código HTML/CSS para incluir o elemento com a informação sobre a contagem. Na Figura 121 encontra-se o resultado de uma execução do programa após esta alteração.

```
function update() {
  const instance = markdown.wasm.instance;

  // Reset elements count.
  instance.exports.reset();

  // Parse the input element value.
  const html = markdown.parse(inputEl.value, {
    parseFlags:
      markdown.ParseFlags.DEFAULT | markdown.ParseFlags.NO_HTML,
  });

  // Update the output element with the parsed html.
  outputEl.innerHTML = html;

  // Update output code highlight.
  updateCodeHighlight();

  // For each element fill the count value.
  document.getElementById("count-bold").innerText =
    instance.exports.get_bold();
  document.getElementById("count-italic").innerText =
    instance.exports.get_italic();
  document.getElementById("count-links").innerText =
    instance.exports.get_link();
  document.getElementById("count-inline-code").innerText =
    instance.exports.get_image();
  document.getElementById("count-images").innerText =
    instance.exports.get_inline_code();
  document.getElementById("count-strikethrough").innerText =
    instance.exports.get_strikethrough();
  const countHeaders = instance.exports.get_headers();
  document.getElementById("count-headers").innerText =
    countHeaders[countHeaders.length - 1];
  document.getElementById("count-tables").innerText =
    instance.exports.get_tables();
  document.getElementById("count-hrs").innerText =
    instance.exports.get_hrs();
  document.getElementById("count-code-blocks").innerText =
    instance.exports.get_code_blocks();
  document.getElementById("count-block-quotes").innerText =
    instance.exports.get_blocks_quotes();
  const countLists = instance.exports.get_lists();
  document.getElementById("count-lists").innerText =
    countLists[countLists.length - 1];
  document.getElementById("count-total").innerText = instance.exports
    .get_count()
    .reduce((accum, cur) => accum + cur, 0);
}
```

Figura 120 - Código JS modificado para suportar a contagem de elementos Markdown

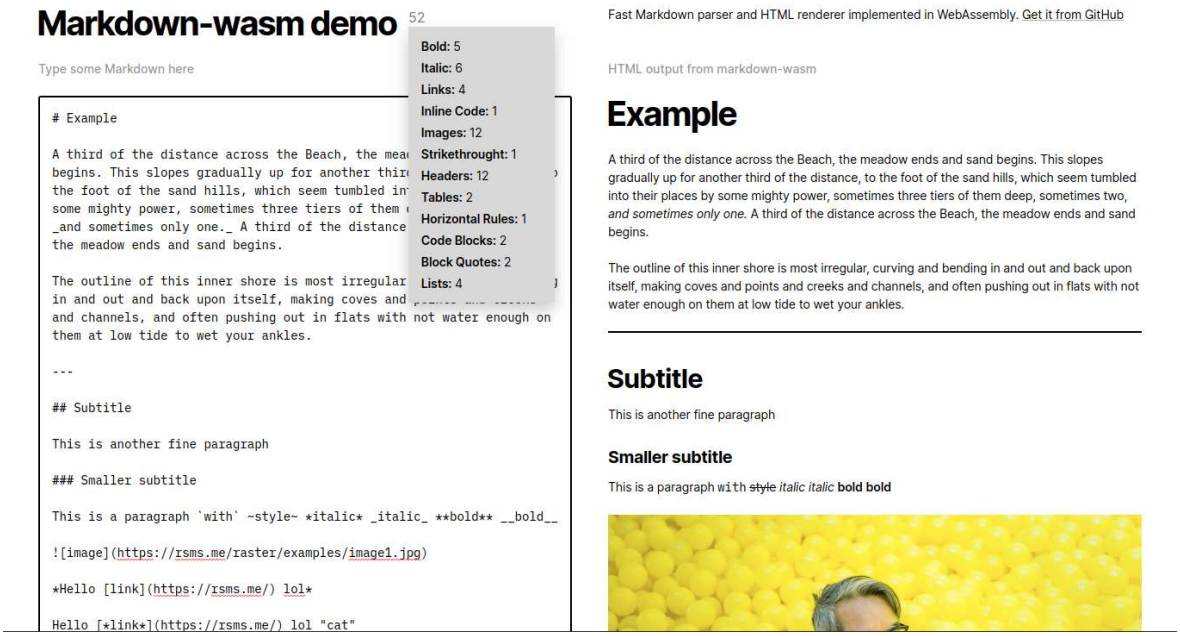


Figura 121 - Resultado da execução do programa *markdown-wasm* com a funcionalidade de contagem

6.4. Jogo Invaders

O último exemplo consiste numa modificação ao jogo dos Invaders. O código original do jogo também se encontra publicado no Github (Battagline, 2021). A modificação consiste na inclusão de dois novos modos de jogo, que são o modo preto e branco, onde as cores são alteradas para tons de cinza, e o modo de velocidade, que permite ao utilizador definir a velocidade com que as naves se movimentam.

Ambos os modos podem ser configurados através de funções importadas que são executadas na função inicial do módulo. Estas configurações são obtidas através dos parâmetros existentes na *query string* da aplicação. Para o primeiro modo foi utilizado um *advice* que através da configuração da função de renderização altera as cores que são imprimidas no elemento canvas do HTML. No que toca à configuração da velocidade, foi necessário um *advice* para alterar o movimento para a esquerda e outro para alterar o movimento para a direita. Ambas as direções estão presentes na mesma função, contudo foi necessário a utilização de *templates* para encontrar cada bloco de código associado. O código implementado para a transformação encontra-se ilustrado na Figura 122. Relativamente às modificações realizadas no código JS, estas encontram-se ilustradas na Figura 123.

```
pointcuts:
  movementFn: () => func(void $f14())
templates:
  add: >
    (local.tee %alien_x%
     (i32.add
      (local.get %val%)
      (i32.const 1)
     )
```

```
    )
  sub: >
    (local.tee %alien_x%
     (i32.sub
      (local.get %val%)
      (i32.const 1)
     )
    )
aspects:
  start: >
    (global.set %speed% (call %get_speed%))
    (global.set %bw% (call %get_bw%))
  context:
    variables:
      speed: i32 = 1
      bw: i32 = 1
    functions:
      get_bw:
        result: i32
        imported:
          module: config
          field: get_bw
      get_speed:
        result: i32
        imported:
          module: config
          field: get_speed
  advices:
    bw_color:
      variables:
        r: i32 = 0
        g: i32 = 0
        b: i32 = 0
        gray: i32 = 0
      pointcut: (i32.param[1] p0) => func(void *(i64, i32, i32, i32, i32, i32))
      order: 0
      advice: >
        (block $BC0
         (br_if $BC0 (i32.eqz (global.get %bw%)))
         (local.set %r%
          (i32.and
           (local.get %p0%)
           (i32.const 0x00_ff_00_00)
          )
         )
         (local.set %g%
          (i32.and
           (local.get %p0%)
           (i32.const 0x00_00_ff_00)
          )
         )
         (local.set %b%
          (i32.and
           (local.get %p0%)
           (i32.const 0xff_00_00_ff)
          )
         )
         (local.set %gray%
          (i32.div_u
```

```
          (i32.add
           (i32.add
            (i32.shr_u (local.get %r%) (i32.const 16))
            (i32.shr_u (local.get %g%) (i32.const 8))
           )
           (local.get %b%)
          )
          (i32.const 3)
         )
        )
        (local.set %p0%
         (local.get %gray%))
        (local.set %p0%
         (i32.add
          (i32.shl (local.get %p0%) (i32.const 8))
          (local.get %gray%)
         )
        )
        (local.set %p0%
         (i32.add
          (i32.shl (local.get %p0%) (i32.const 8))
          (local.get %gray%)
         )
        )
       )
       %this%
  right:
    pointcut: () => movementFn() && template(add)
    advice: >
      (local.tee %add:select(alien_x)%
       (i32.add
        (local.get %add:select(val)%)
        (global.get %speed%)
       )
      )
  left:
    pointcut: () => movementFn() && template(sub)
    advice: >
      (local.tee %sub:select(alien_x)%
       (i32.sub
        (local.get %sub:select(val)%)
        (global.get %speed%)
       )
      )
```

Figura 122 - Código da transformação com os novos modos de jogo no Invaders

```
// get the mode value
const mode = getParameterByName('mode');
// get the speed value
const speed = getParameterByName('speed');
const importObject = {
    (...)
    config: {
        // add the getters for the module
        get_bw: () => mode === 'bw',
        get_speed: () => +speed || 1
    }
};

// (...)
```

Figura 123 - Código JS modificado para suportar os novos modos de jogo no Invaders

6.5. Discussão da Validação

A validação realizada teve por base a criação e exploração de vários cenários típicos de utilização que funcionam como provas de conceito que sustentam a aplicabilidade da linguagem e ferramenta desenvolvida.

No seu conjunto, este processo permitiu demonstrar que a expressividade da linguagem é suficiente para descrever casos de uso considerados representativos de utilizações potenciais da ferramenta, provando-se assim que é possível:

- a instrumentação do código através da inserção de logs em pontos definidos pelo utilizador. Conforme os exemplos das Secções 6.1.1 e 6.2.6.1, o utilizador pode optar por implementar um *log* simples apenas com recurso a modificações do WASM, como também pode optar por elaborar *logs* mais detalhados com recurso ao JS auxiliar e tipos adicionais.
- a monitorização de desempenho de atividades definidas pelo utilizador (Secções 6.1.2 e 6.2.2). Esta monitorização também poderá recorrer a tipos adicionais para identificar as atividades que se pretendem monitorizar (Secção 6.2.6.1). Na Secção 6.2.5.1 é demonstrada outra forma de monitorizar diferentes atividades sem a necessidade de recurso ao JS auxiliar.
- a manipulação do fluxo de execução. Nos exemplos das Secções 6.1.3 e 6.1.4 foi alterado o fluxo de execução para que fosse integrado no programa um novo algoritmo com melhor desempenho, cuja execução alternaria com o algoritmo já existente de acordo com uma configuração do utilizador.
- a aplicação de otimizações de desempenho. Foi implementado nos exemplos das Secções 6.2.3 e 6.2.6.2 um mecanismo de *cache* que permitiu reduzir consideravelmente o tempo de execução da respetiva atividade. Com recurso aos tipos adicionais ao WASM e as transformações *runtime*, foi possível criar uma transformação transversal a múltiplas atividades, onde o mecanismo se adapta ao contexto onde é inserido. Nas Secções 6.2.5.2 e 6.2.5.3 foram também

implementadas duas otimizações, a remoção de chamadas e o *loop unrolling*, com recurso à pesquisa por padrão disponibilizada pela ferramenta.
- a reparação de erros. A Secção 6.2.4 demonstra um exemplo com a aplicação de uma transformação que visa mitigar um erro detetado durante a execução do programa.
- a extensão das funcionalidades originais. Nas Secções 6.2.1, 6.2.5, 6.3 e 6.4 foram implementadas transformações que visam estender a funcionalidade dos programas originais ao inserir e integrar novas funcionalidades.

Foi também demonstrado que o comportamento da ferramenta cumpre os requisitos estabelecidos, sendo estes:

- **Adicionar transformação em vários pontos específicos do programa** – para todas as transformações aplicadas nos exemplos apresentados foi necessário definir um ponto específico do código, onde depois foram inseridas as alterações.
- **Adicionar elementos ao programa** – diversas transformações aplicadas nos exemplos apresentados possuem a inclusão de novos elementos no programa, tais como, funções, variáveis globais, etc.
- **Aceder aos elementos criados no código** – os elementos criados nas transformações foram constantemente acedidos nos vários exemplos apresentados. Este acesso tanto foi feito com recurso às *static expressions* como às *runtime expressions*.
- **Aceder ao contexto do ponto onde será incluída a transformação** – nos exemplos apresentados, existem transformações onde é feito o acesso ao contexto fornecido pelo *pointcut*.
- **Manipular dados disponibilizados na transformação** – através das funções de transformação disponibilizadas pelas *static expression*s, diversos exemplos realizam a manipulação dos dados acedidos.
- **Executar transformações sensíveis ao contexto da execução** – a utilização das *runtime expressions* nos vários exemplos apresentados permitiu que as transformações fossem sensíveis ao contexto.
- **Permitir a utilização de tipos complexos** – os tipos complexos consistem nos tipos adicionais à especificação do WASM, e são compostos pelos tipos: *string*, *array* e mapa. Todos estes tipos encontram-se implementados nos exemplos apresentados.

Por fim, os resultados apresentados para cada uma das transformações durante este processo permitiu a verificação funcional da ferramenta, de acordo com a especificação. Isto significa que, se a especificação das transformações estiver correta, o programa resultante da sua aplicação é o esperado pelo utilizador.

CAPÍTULO 7.
CONCLUSÃO

O trabalho desenvolvido para este documento consiste numa ferramenta de manipulação de código WASM, chamada WasmManipulator, que permite a aplicação de transformações estáticas ou *runtime* ao código. A definição das transformações é baseada no paradigma orientado a aspetos. Ao usufruir das várias características deste paradigma, tais como modularização, reutilização, simplicidade e legibilidade, permitiu a criação de uma linguagem flexível e poderosa que possibilita a realização de transformações complexas no código WASM.

Estas transformações apoiam-se na linguagem JS para que sejam suportadas funcionalidades adicionais à especificação do WASM. Esta dependência faz todo o sentido, uma vez que de um lado temos o JS, que se resume à tecnologia principal no desenvolvimento de aplicações dinâmicas *web*, e do outro temos o WASM, que é completamente interoperável com JS, e que permite a melhoria no desempenho destas aplicações, possibilitando assim o desenvolvimento de aplicações que anteriormente não era possível na *web*, tais como, edição de vídeo, renderização 3D, videojogos, etc.

7.1. Resumo do Trabalho

O processo de desenvolvimento da ferramenta iniciou com uma fase de planeamento, onde se começou poranalisar um conjunto de ferramentas para o WASM que permitem a manipulação e transformação do código. Nesta análise tentou-se perceber até que ponto as ferramentas que atualmente existem no mercado permitem realizar uma transformação simples e flexível do código. Contudo, não foi encontrada nenhuma que permitisse atingir o nível de transformações pretendido para resolver o problema exposto inicialmente.

Sabendo isto, a análise foi expandida para incluir ferramentas utilizadas para outras linguagens de programação. Nesta análise foram encontradas diversas ferramentas válidas para o que se pretendia, contudo, destacou-se uma que, utilizando os conceitos do paradigma orientado a aspetos, permite realizar quase todo o tipo de transformações de código, mantendo uma forma organizada e simples. Desta forma, os conceitos abordados nesta ferramenta (conceitos do paradigma orientado a aspetos) foram considerados para ser a base das transformações de uma possível ferramenta semelhante para o WASM.

Como um dos objetivos era também permitir a pesquisa por padrão, foram analisadas algumas ferramentas de pesquisa de código que podiam ser integradas numa possível ferramenta WASM, juntamente com os conceitos do paradigma orientado a aspetos. Uma das ferramentas analisadas adequou-se perfeitamente, pois não só é simples e eficiente, como também possui total compatibilidade com o código WAT.

Depois desta análise conclui-se que faltava no ecossistema do WASM uma ferramenta que permitisse a transformação simples e flexível do código, tal como já existe para outras linguagens mais maduras. Com isto, optou-se por iniciar o desenvolvimento de uma ferramenta que, baseando-se numa ferramenta de manipulação de código Java, implementou

os mesmos conceitos do paradigma orientado a aspetos para permitir a transformação de código WASM de uma forma flexível, poderosa e organizada.

Procedeu-se à fase de planeamento da ferramenta WasmManipulator. Esta fase iniciou com a elaboração dos requisitos que a ferramenta deveria cumprir. Em suma, foram estabelecidos os seguintes requisitos:

- Adicionar transformação em vários pontos específicos do programa.
- Adicionar elementos ao programa.
- Aceder aos elementos criados no código.
- Aceder ao contexto do ponto onde será incluída a transformação.
- Manipular dados disponibilizados na transformação.
- Executar transformações sensíveis ao contexto da execução.
- Permitir a utilização de tipos complexos.
- Executar sem a interação do utilizador.

Com os requisitos definidos, iniciou-se a projeção da linguagem que seria interpretada pela ferramenta para executar as devidas transformações. Esta linguagem recorreu ao formato YAML para estruturar a definição dos vários elementos existentes. Para cada elemento existe uma sintaxe específica, detalhada no presente documento.

O próximo passo foi estudar as tecnologias que seriam utilizadas para o desenvolvimento desta ferramenta, o WasmManipulator. Como o tipo de ferramenta pretendido seria uma aplicação executada no terminal, optou-se por escolher uma linguagem de programação simples e eficiente, o Go. Para que o conteúdo fosse convertido de WASM para WAT e vice-versa, foi utilizada a ferramenta WABT. A pesquisa por padrão foi implementada com recurso a uma das ferramentas analisadas na fase anterior, o Comby. Por fim, o código JS gerado pela ferramenta é otimizado através da ferramenta MinifyJS.

Com as tecnologias já escolhidas, procedeu-se à análise da arquitetura e fluxo a implementar. De uma forma resumida, a arquitetura baseou-se na divisão da lógica em *packages* que interagem entre si. Cada *package* encapsula a lógica associada aos diversos componentes da ferramenta, tais como, *parse* da linguagem de transformação, obtenção de *join-points*, geração de código, etc. O fluxo de execução pode ser dividido em quatro fases principais, a configuração, a leitura dos ficheiros de entrada, a transformação do módulo, e a geração de resultados. Toda a informação relativa à arquitetura e fluxo de execução encontra-se devidamente detalhada no presente documento.

Seguiu-se a fase de desenvolvimento onde foi implementada a ferramenta de acordo com o planeado. Nesta fase, o desenvolvimento foi realizado de forma progressiva, onde por vezes foram ajustados os requisitos iniciais, uma vez que a ferramenta à medida que ia sendo implementada, também ia sendo testada e validada com pequenos exemplos.

Depois de implementada, de forma a demonstrar a sua aplicabilidade preparou-se um conjunto de exemplos, presentes neste documento, com as transformações mais comuns deste tipo de linguagens. Os exemplos definidos têm como objetivo retratar contextos reais,

tendo sempre por base, código publicado. Ao implementar com sucesso essas transformações, verificou-se que a ferramenta é útil para ser utilizada em projetos de caráter real, sendo capaz de realizar transformações bastante complexas, de uma forma organizada e relativamente simples. Desta forma, foi possível validar que a ferramenta é capaz de:

- instrumentar o código;
- realizar otimizações de desempenho complexas, tais como, a implementação de *caching* e *loop unrolling*;
- analisar e reparar erros de execução;
- analisar e manipular o fluxo de execução;
- monitorizar o desempenho;
- estender as funcionalidades originais.

7.2. Contribuições

As principais contribuições deste trabalho são:

- Especificação de uma linguagem orientada a aspetos para manipulação de código WebAssembly e Javascript.
- Implementação e disponibilização de ferramenta de manipulação de WebAssembly e Javascript baseada na linguagem anterior.
- Criação de documentação com exemplos de aplicação de linguagem em cenários típicos de utilização.
- Submissão, durante o mês de fevereiro, de artigo para conferência.

7.3. Trabalho Futuro

A ferramenta desenvolvida neste trabalho poderá ser sujeita a melhorias adicionais.. Desta forma, propõe-se como trabalho futuro:

- Realizar a compilação do módulo e as respetivas transformações em *streaming*. Desta forma, o módulo WASM não necessitava de estar armazenado internamente na ferramenta, permitindo a transformação de código com dimensões muito superiores. Para além disso, a transformação também seria muito mais eficiente, uma vez que só era feito a análise e transformação das secções que precisavam ser modificadas.
- Implementar uma otimização no código JS associado às expressões *runtime*. O código das expressões passado para o módulo JS é devidamente dividido num *array* de inteiros *32-bit*. Cada elemento do *array* é depois enviado para o JS através da chamada a uma função importada do JS. Como consequência, quanto maior for a dimensão da expressão, maior será o número de chamadas consecutivas a essa função. Desta forma, ao aplicar uma otimização ao código da expressão, utilizando por exemplo a ferramenta MinifyJS, reduziria substancialmente o número de instruções associadas.

- Adicionar uma camada extra que permita a definição de código mais alto-nível na especificação da transformação, como por exemplo, o C/C++ ou AssemblyScript, ao invés da utilização direta do código WAT. O código de alto-nível definido, depois seria devidamente convertido para o respetivo WAT.
- Suportar a manipulação dos restantes elementos do módulo, tais como, tabelas e memórias lineares.

REFERÊNCIAS BIBLIOGRÁFICAS

Andersson, R. (11 de 2021). *markdown-wasm*. Obtido de GitHub: https://github.com/jotar910/markdown-wasm

AspectJ, T. (2001). The AspectJ Programming Guide. Em t. A. Team, *Development Aspects.* Xerox Corporation. Obtido de Eclipse: https://www.eclipse.org/aspectj/doc/released/progguide/examples-development.html

Atapattu, S. (03 de 2020). *Bringing You Up to Speed on How Compiling WebAssembly is Faster.* Obtido de CS 6120: https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/wasm/

Baker, J. (2019). *Hacking WebAssembly Games with Binary Instrumentation*. Obtido de TIB AV-Portal: https://av.tib.eu/media/48379

Battagline, R. (11 de 2021). *WAT-Invaders*. Obtido de GitHub: https://github.com/battlelinegames/WAT-Invaders

Bebenita, M. (03 de 2018). *WebAssembly Explorer*. Obtido de GitHub: https://mbebenita.github.io/WasmExplorer/

Ben-Kiki, O., Evans, C., & Net, I. d. (07 de 2009). *YAML Ain't Markup Language (YAML) Version 1.2.* Obtido de YAML Ain't Markup Language (YAML) Version 1.2: https://yaml.org/spec/1.2.0/

Bohm, W. (2014). *Abstract Syntax tree (AST) - Visitor patterns* . Obtido de CS 453 Introduction to Compilers: https://www.cs.colostate.edu/~cs453/yr2014/

Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. (2008). *Stratego/XT 0.17. A language and toolset for program transformation.* Association for Computer Machinery.

Bruneton, E. (2011). *ASM 4.0 A Java bytecode engineering library.*

Bryant, D. (2017). *Why WebAssembly is a game changer for the web — and a source of pride for Mozilla and Firefox.* Obtido de Medium: https://medium.com/mozilla-tech/why-webassembly-is-a-game-changer-for-the-web-and-a-source-of-pride-for-mozilla-and-firefox-dda80e4c43cb

Burke, E. (2001). XPath Basics. Em E. Burke, *Java and XSLT.* O'Reilly & Associates. Obtido de docstore.mik.ua/orelly/xml/jxslt/ch02_04.htm

Can I Use. (11 de 2021). *WebAssembly*. Obtido de Can I Use: https://caniuse.com/wasm

Chernysh, V. (09 de 2015). *ActiveX in .NET Applications*. Obtido de Code Project: https://www.codeproject.com/Tips/1028587/ActiveX-in-NET-Applications

Clark, L. (10 de 2018). *Calls between JavaScript and WebAssembly are finally fast.* Obtido de Hacks Mozilla: https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%F0%9F%8E%89/

Cneude, M. (01 de 2018). *The DRY Principle: Benefits and Costs with Examples*. Obtido de The Valuable Dev: https://thevaluable.dev/dry-principle-cost-benefit-example/

Comby. (2021). *Comby*. Obtido de Comby Docs: https://comby.dev/docs/overview

Community, W. (02 de 2020). *FAQ.* Obtido de GitHub: https://github.com/WebAssembly/design/blob/main/FAQ.md

Community, W. F. (2021). *Wallpaper Flare*. Obtido de Wallpaper Flare: wallpaperflare.com

Contributors, L. (2021). *Logrus*. Obtido de GitHub: https://github.com/sirupsen/logrus

Contributors, P. (2021). *Participle*. Obtido de GitHub: https://github.com/alecthomas/participle

Contributors, P. (2021). *PFlag*. Obtido de GitHub: https://github.com/spf13/pflag

Contributors, V. (2021). *Viper*. Obtido de GitHub: https://github.com/spf13/viper

Contributors, W.-J.-T. (2021). *WASM-JSON-Toolkit*. Obtido de GitHub: https://github.com/ewasm/wasm-json-toolkit

Costa, E. D., & Berkenbrock, G. R. (2003). *Programação Orientada a Aspectos.*

Dambekalns, K., Müller, C., Lemke, R., & Waidelich, B. (09 de 2021). *Aspect-Oriented Programming.* Obtido de Flow Framework: https://flowframework.readthedocs.io/en/stable/TheDefinitiveGuide/PartIII/AspectOrientedProgramming.html

Doherty, E. (2020). *What is Object Oriented Programming? OOP Explained in Depth*. Obtido de Educative: https://www.educative.io/blog/object-oriented-programming

Erik. (12 de 2018). *YAML Tutorial: Everything You Need to Get Started in Minutes.* Obtido de CloudBees: https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started

Fioretti, M. (03 de 2021). *WebAssembly Security, Now and in the Future.* Obtido de The Linux Foundation: https://training.linuxfoundation.org/blog/webassembly-security-now-and-in-the-future/

Foundation, E. (2021). *The AspectJ Project*. Obtido de Eclipse: http://eclipse.org/aspectj

Fredriksson, S. (2020). *WebAssembly vs. its predecessors.*

Gear Technologies. (10 de 2021). *What is the WebAssembly Virtual Machine & Why should you use it?* Obtido de Medium: https://medium.com/@gear_techs/what-is-the-webassembly-virtual-machine-why-should-you-use-it-5bfa521e7880

Gifford, C. (2019). *Design and Analysis of an Instrumenting Profiler for Webassembly.* Cal Poly.

Gohman, D., Lepesme, J.-B., Qwerty2501, Spencer, O., & Um, J. (01 de 2021). *WebAssembly Reference Manual.* Obtido de GitHub: https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md

Hernandez, L. (2021). *What Is WebAssembly And What Is It Used For?* Obtido de FullStack Labs: https://www.fullstacklabs.co/blog/what-is-webassembly

ICHI.PRO. (2021). *001 - WebAssembly e JavaScript: A introdução.* Obtido de ICHI.PRO: https://ichi.pro/pt/001-webassembly-e-javascript-a-introducao-245031558582613

ISEC. (2021). *Apresentação*. Obtido de ISEC - Intituto Superior de Engenharia de Coimbra: https://www.isec.pt/

Jangda, A., Power, B., E. D., & Guha, A. (2019). *Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code.*

JBoss-Javassist. (2020). *Javassist*. Obtido de Javassist by jboss-javassist: https://www.javassist.org/

Johnson, B., & Simha, R. (2019). *CRAQL: A Composable Language for Querying Source Code.* Cornell University.

Kanjilal, J. (03 de 2016). *My two cents on aspect-oriented programming.* Obtido de InfoWorld: https://www.infoworld.com/article/3040557/my-two-cents-on-aspect-oriented-programming.html

Kel. (01 de 2017). *Why not java or .net?* Obtido de GitHub: https://github.com/WebAssembly/design/issues/960

Krill, P. (06 de 2017). *WebAssembly wins! Google pulls plug on PNaCl.* Obtido de InfoWorld: https://www.infoworld.com/article/3199191/webassembly-wins-google-pulls-plug-on-pnacl.html

Kukunas, J. (2015). Loop Unrolling. Em J. Kukunas, *Power and Performance.* Elsevier Inc.

Lab, S. (2020). *Wasabi*. Obtido de Wasabi Software Lab: http://wasabi.software-lab.org/

Laddad, R. (2010). *AspectJ in Action.* Manning Publications Co.

Lehmann, D., & Pradel, M. (2018). *Wasabi: A Framework for Dynamically Analyzing.* Cornell University.

Long, J. (2021). *Javassist by jboss-javassist*. Obtido de Javassist: http://www.javassist.org/

Math, D. (2012). *How does the calculator find values of sine (or cosine or tangent)?* Obtido de Home School Math: https://www.homeschoolmath.net/teaching/sine_calculator.php

MDN Contributors, M. (2021). *ArrayBuffer.* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer

MDN Contributors, M. (2021). *Arrow function expressions*. Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

MDN Contributors, M. (2021). *console.log().* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/API/Console/log

MDN Contributors, M. (2021). *eval().* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

MDN Contributors, M. (2021). *Understanding WebAssembly text format.* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

MDN Contributors, M. (2021). *Using Fetch.* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

MDN Contributors, M. (2021). *WebAssembly.* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly

MDN Contributors, M. (2021). *WebAssembly Concepts*. Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

MDN Contributors, M. (2021). *WebAssembly.instantiate().* Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate

MDN Contributors, M. (2021). *WebAssembly.instantiateStreaming()*. Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiateStreaming

MDN Contributors, M. (2021). *XPath*. Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/XPath

MDN Contributors, M. (2021). *XSLT*. Obtido de MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/XSLT

Mihajlija, M. (2019). *WebAssembly: How and why*. Obtido de LogRocket: https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71

Net, I. d. (10 de 2021). *YAML 1.2*. Obtido de The Official YAML Web Site: https://yaml.org/

Nikolaev, A. (2021). *More binary trees. S-expressions.* Obtido de GitHub: https://a-nikolaev.github.io/fp/lec/9/

Noleto, C. (04 de 2020). *Programação Funcional: o que é, principais conceitos e vantagens!* Obtido de Trybe: https://blog.betrybe.com/tecnologia/programacao-funcional/

Optasy. (2018). *WebAssembly vs JavaScript: Is WASM Faster than JS? When Does JavaScript Perform Better?* Obtido de Medium: https://medium.com/@OPTASY.com/webassembly-vs-javascript-is-wasm-faster-than-js-when-does-javascript-perform-better-db86d2ecf2cc

Pike, R. (08 de 2011). *Lexical Scanning in Go.* Obtido de Talks Golang: https://talks.golang.org/2011/lex.slide

Qwokka, J. (2019). *WebAssembly Intrumentation Library (WAIL)*. Obtido de GitHub: https://github.com/Qwokka/WAIL

Rakshit, S. (09 de 2019). *What is the inline function in JavaScript ?* Obtido de GeeksforGeeks: https://www.geeksforgeeks.org/what-is-the-inline-function-in-javascript/

Restivo, A. (2006). *The Case for Aspect Oriented Programming.* ResearchGate.

Rodrigues, J. (12 de 2021). *WasmManipulator*. Obtido de GitHub: https://github.com/jotar910/wasm-manipulator

Rossberg, A. (2021). *WebAssembly Specification.* WebAssembly Community Group.

Sendilkumarn. (01 de 2020). *Understanding WebAssembly Text Format - From WTF to WAT.* Obtido de sendilkumarn.com: https://sendilkumarn.com/blog/webassembly-text-format/

Silva, R. (11 de 2014). *Programação orientada a aspectos.* Obtido de SlideShare: https://www.slideshare.net/blenicio/programao-orientada-a-aspectos-41758411

Sletten, B. (2021). WebAssembly Memory. Em B. Sletten, *WebAssembly: The Definitive Guide.* O'Reilly Media, Inc.

Smith, B., & Community, W. (10 de 2021). *WABT: The WebAssembly Binary Toolkit*. Obtido de GitHub: https://github.com/WebAssembly/wabt

Spinczyk, O., & GmbH, p.-s. (2021). *Documentation: AspectC++ Language Reference.*

Team, A. (2021). *AsmJS*. Obtido de asm.js: http://asmjs.org/

Team, C. (2021). *C++*. Obtido de Cplusplus: https://www.cplusplus.com/

Team, D. (2019). *Pros and Cons of JavaScript – Weigh them and Choose wisely!* Obtido de DataFlair : https://data-flair.training/blogs/advantages-disadvantages-javascript/

Team, G. (09 de 2021). *Expression Evaluation*. Obtido de GeekforGeeks: https://www.geeksforgeeks.org/expression-evaluation/

Team, G. (2021). *Go*. Obtido de Go: https://go.dev/

Team, G. (09 de 2021). *HeapSort*. Obtido de GeeksforGeeks: https://www.geeksforgeeks.org/heap-sort/

Team, G. (05 de 2021). *StoogeSort*. Obtido de GeeksforGeeks: https://www.geeksforgeeks.org/stooge-sort/

Team, L. A. (2021). *LLVM*. Obtido de The LLVM Compiler Infrastructure: https://llvm.org/

Team, M. (2021). *C# documentation*. Obtido de Microsoft: https://docs.microsoft.com/en-us/dotnet/csharp/

Team, M. (2021). *CLR overview*. Obtido de Microsoft: https://docs.microsoft.com/en-us/dotnet/standard/clr

Team, O. (2004). Aspect-oriented software engineering. Em O. Team, *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley Professional.

Team, O. (2021). *Java*. Obtido de Java: https://www.java.com/

Team, O. (2021). *Java Applets*. Obtido de The Java Tutorials: https://docs.oracle.com/javase/tutorial/deployment/applet/index.html

Team, O. (2021). *JVM*. Obtido de Java: https://www.java.com/pt-BR/download/manual.jsp

Team, R. (2021). *Rust*. Obtido de RustLang: https://www.rust-lang.org/

Team, T. (2006). *String*. Obtido de TechTerms.com: https://techterms.com/definition/string

Team, T. (2021). *TypeScript*. Obtido de TypeScript: https://www.typescriptlang.org/

Team, W. (2021). *SQL Tutorial*. Obtido de W3Schools: https://www.w3schools.com/sql/

Team, W. (09 de 2021). *XPath Tutorial.* Obtido de W3Schools: https://www.w3schools.com/xml/xpath_intro.asp

Tomassetti, G. (09 de 2017). *A Guide to Parsing: Algorithms and Terminology.* Obtido de Strumenta: https://tomassetti.me/guide-parsing-algorithms-terminology/

Tulka, T. (08 de 2021). *Imaging with WebAssembly in AssemblyScript*. Obtido de GitHub: https://github.com/ttulka/assemblyscript-samples/tree/main/imaging

Wang, D., Knutson, S., Donor, A., Botting, D., & Harris, D. (11 de 2014). *Aspect Oriented Programming.* Obtido de WikiWikiWeb: https://wiki.c2.com/?AspectOrientedProgramming

Wasmtime. (November de 2021). *Wasmtime.* Obtido de Wasmtime Docs: https://docs.wasmtime.dev/print.html

WebAssembly. (2017). *WebAssembly*. Obtido de WebAssembly: https://webassembly.org/

ANEXOS

Anexo A. Opções de Configuração 153
Anexo B. Especificação Detalhada da Linguagem 157
1. Sintaxe 158
1.1. String 159
1.2. Identifier 159
1.3. Type 159
1.4. Variable 160
1.5. Code 160
1.5.1. CodeFunction 161
1.5.2. CodeAdvice 161
1.6. Pointcut 161
1.6.1. PointcutGlobal 161
1.6.2. PointcutAdvice 161
1.7. Template 162
2. Pointcut Expressions 162
2.1. Pointcut func 163
2.1.1. Sintaxe 163
2.1.2. Dados de Contexto 165
2.2. Pointcut call 166
2.2.1. Sintaxe 166
2.2.2. Dados de Contexto 166
2.3. Pointcut args 167
2.3.1. Sintaxe 167
2.3.2. Dados de Contexto 167
2.4. *Pointcut* returns 167
2.4.1. Sintaxe 167
2.4.2. Dados de Contexto 167
2.5. *Pointcut* template 168
2.5.1. Sintaxe 168
2.5.2. Dados de Contexto 168
7.3.1.1. Operadores Lógicos 168

3. Code Expressions 169
3.1. Static Expressions 169
3.1.1. Sintaxe 169
3.1.2. Contexto 169
3.1.3. Tipos de Variáveis 170
3.1.4. Funções de Transformação 170
3.1.5. Palavras Reservadas (*Keywords*) 176
3.1.6. Observações 176
3.2. Runtime Expressions 177
3.2.1. Runtime References 178
3.2.2. Palavras Reservadas (*Keywords*) 179
3.3. Template Expressions 179
3.3.1. Template Keywords 179
3.3.2. Funcionamento 180
4. Modo Inteligente (Smart) 181

ANEXO A. OPÇÕES DE CONFIGURAÇÃO

Ficheiro de entrada do módulo

Nesta configuração o utilizador deve inserir o ficheiro de entrada que pretende modificar. O ficheiro pode encontrar-se no formato binário (.wasm) ou no formato textual (.wat), e deve conter um módulo WASM válido.

Exemplos:

- ./wmr --data_dir="$HOME/" --in_module="data/module.wasm" (o ficheiro encontra-se em $HOME/data/module.wasm)
- WMR_IN_MODULE="data/module.wasm" ./wmr (o ficheiro encontra-se em ./data/module.wasm)

Ficheiro de entrada da transformação

A configuração do ficheiro de entrada que contém as instruções de transformação do módulo deve assumir o formato YAML (YAML Ain't Markup Language).

Exemplos:

- ./wmr --in_transform="data/transf.yml" (o ficheiro encontra-se em ./data/transf.yml)
- WMR_IN_TRANSFORM="data/transf.yml" ./wmr (o ficheiro encontra-se em ./data/transf.yml)

Ficheiro de saída do módulo transformado

Esta configuração representa o ficheiro resultante das transformações. Este ficheiro será constituído por um módulo válido no formato binário.

Exemplos:

- ./wmr --out_module="result.wasm" (o ficheiro encontra-se em ./result.wasm)
- WMR_OUT_MODULE="result.wasm" ./wmr (o ficheiro encontra-se em ./result.wasm)

Ficheiro de saída do JS auxiliar

O ficheiro de saída auxiliar ao módulo transformado assume o formato JS (.js). A menos que a configuração "Gerar sempre o ficheiro JS" esteja ativa, este ficheiro nem sempre será criado após a execução da ferramenta. Isto deve-se ao facto do código JS apenas ser necessário caso o utilizador utilize tipos complexos ou *runtime expressions* na transformação do módulo.

Exemplos:

- ./wmr --out_js="result.js" (o ficheiro encontra-se em ./result.js)
- WMR_OUT_JS="result.js" ./wmr (o ficheiro encontra-se em ./result.js)

Ficheiro de saída do módulo original

Nesta configuração é indicado se o ficheiro binário inicial no qual sofreu as transformações deverá ser criado. Este ficheiro é criado caso a configuração esteja definida corretamente e o ficheiro de entrada com o módulo a ser transformado seja do tipo textual (.wat).

Exemplos:

- ./wmr --out_module_orig="module_orig.wasm" (o ficheiro encontra-se em ./module_orig.wasm)
- WMR_OUT_MODULE_ORIG="module_orig.wasm" ./wmr (o ficheiro encontra-se em ./module_orig.wasm)

Diretoria com as dependências

Consiste no caminho base (absoluto ou relativo) onde se encontram as dependências necessárias para a execução. A partir desse caminho, devem existir os seguintes executáveis:

- ${WMR_DEPENDENCIES_DIR}/wabt/wasm2wat
- ${WMR_DEPENDENCIES_DIR}/wabt/wat2wasm
- ${WMR_DEPENDENCIES_DIR}/minifyjs/bin/minify.js
- ${WMR_DEPENDENCIES_DIR}/comby/comby

Por predefinição o caminho para as dependências é "./dependencies", sendo que inicia na diretoria onde foi executada a ferramenta.

Esta configuração pode ser ignorada caso o utilizador possua os executáveis definidos na variável de ambiente "PATH". Esta opção não é recomendada, uma vez que a versão instalada pode causar problemas na execução da ferramenta.

Exemplos:

- ./wmr --dependencies_dir="$HOME/"
- WMR_DEPENDENCIES_DIR="$HOME/" ./wmr

Diretoria com dados para execução

Consiste no caminho (absoluto ou relativo) onde se encontram os dados de entrada necessários à ferramenta, e onde os resultados serão armazenados após terminar a execução.

Por predefinição o caminho da diretoria dos dados de entrada é o mesmo que a diretoria onde foi executada a ferramenta.

Exemplos:

- ./wmr --data_dir="$HOME/"
- WMR_DATA_DIR="$HOME/" ./wmr

Ficheiro de logs

O ficheiro de logs é uma configuração opcional, sendo que por predefinição o logs são imprimidos para a consola. Caso a configuração esteja definida com um ficheiro válido, os

logs são imprimidos nesse ficheiro, e não para a consola. Atenção, ficheiros com o mesmo nome serão completamente substituídos por este novo.

Exemplos:

- ./wmr --log_file="logs" (o ficheiro encontra-se em ./logs)
- WMR_LOG_FILE="logs" ./wmr (o ficheiro encontra-se em ./logs)

Incluir *advices*

Esta configuração recebe um array com os nomes dos *advices* que devem ser incluídos na transformação. Esta filtragem permite ao utilizador aplicar apenas os *advices* pretendidos, e assim obter resultados diferentes, para o mesmo ficheiro de transformação.

Exemplos:

- ./wmr --include=advice_1,advice_2
- WMR_INCLUDE=advice_1,advice_2 ./wmr

Excluir *advices*

Esta configuração recebe um array com os nomes dos *advices* que devem ser excluídos na transformação. Esta filtragem permite ao utilizador remover *advices* indesejados, e assim obter resultados diferentes, para o mesmo ficheiro de transformação.

Quando definida juntamente com a configuração "Incluir *advices"*, a remoção dos *advices* é feita com base nos que são resultantes dessa configuração.

Exemplos:

- ./wmr --exclude=advice_1,advice_2
- WMR_EXCLUDE=advice_1,advice_2 ./wmr

Gerar sempre o ficheiro JS

Quando esta configuração está ativa, o ficheiro auxiliar JS é sempre criado após a execução da ferramenta, independentemente se é necessário para a integração do módulo WASM numa aplicação ou não.

Por predefinição esta configuração está inativa, sendo que, caso seja necessário para o resultado da ferramenta, o JS é gerado.

Exemplos:

- ./wmr --print_js
- WMR_PRINT_JS=true ./wmr

Gerar sempre o módulo transformado

Ao ativar esta configuração, o módulo resultante da transformação é sempre gerado. Isto significa que a execução dar-se-á mesmo que não sejam encontrados *join-points* no módulo para os *advices* definidos.

Apesar do módulo resultante não possuir qualquer transformação ao código existente, este pode conter código novo inserido pelo utilizador. Este código pode ser inserido através de variáveis globais ou novas funções que interagem entre si ou com os elementos existentes no módulo original (apenas com recurso a índices numéricos).

Exemplos:

- ./wmr --allow_empty
- WMR_PRINT_JS=true ./wmr

Imprimir todos os logs

Esta configuração permite que logs de rastreamento sejam imprimidos juntamente com os restantes logs. Com isto, é possível fornecer ao utilizador logs mais detalhados sobre a execução da aplicação. Quando desativa, apenas imprime logs do tipo *info*, ou seja, logs menos detalhados.

Exemplos:

- ./wmr --print_js
- WMR_PRINT_JS=true ./wmr

Não ordenar advices

Indica se deve ou não ordenar os advices segundo o campo "Order". Por predefinição os *advices* são ordenados, sendo que caso não seja indicado o valor no campo "Order", o *advice* é colocado no fim da lista de execução.

Exemplos:

- ./wmr --ignore_order
- WMR_IGNORE_ORDER=true ./wmr

Nota:

Qualquer caminho inserido será relativo à diretoria com os dados para execução, isto é, o caminho terá por base o caminho definido na configuração “Diretoria com dados para execução”.

ANEXO B. ESPECIFICAÇÃO DETALHADA DA LINGUAGEM

A linguagem para transformação de WASM utiliza o YAML para organizar e estruturar as instruções. Com isto, a sua estrutura de campos encontra-se ilustrada na Figura 124. À frente de cada campo existe um breve descrição dos mesmos no formato de um comentário.

```
{
  Pointcuts: Mapa, // possui a definição de pointcuts globais, isto é, que pode
rão ser utilizados em qualquer advice definido.
  Aspects: Mapa, // possui os dados de transformação do módulo.
  Context: { // definição e inicialização de elementos no contexto global do mó
dulo.
    Variables: Mapa, // declaração e inicialização das variáveis globais.
    Functions: { // definição das funções a acrescentar ao módulo.
        Variables: Mapa, // declaração e inicialização das variáveis locais.
        Args: Array<{ // define a lista de argumentos recebidos pela função.
          Name: string, // referente ao nome do argumento.
          Type: string, // referente ao tipo do argumento.
        }>,
        Result: string, // tipo do valor retornado pela função.
        Code: string, // código da função. O código deve ter o formato WAT e po
de possuir expressões específicas da aplicação.
        Imported: { // declara a função como sendo uma função importada.
          Module: string, // nome do módulo onde a definição da função está ins
erida.
          Field: string, // nome do campo onde a definição da função está inser
ida (dentro do módulo).
        },
        Exported: string, // declara a função como sendo uma função exportada.
Caso a função já tenha sido marcada como uma função importada, esta instrução é
 ignorada.
    },
  },
  Advices: { // definição dos advices a utilizar na transformação.
    Pointcut(obrigatório): string, // definição do pointcut para o advice. Poin
tcuts globais podem ser utilizados aqui.
    Variables: Mapa, // declaração e inicialização das variáveis locais a inser
ir nas funções a aplicar as alterações.
    Advice: string, // código que substituirá os join-
points. O código deve ter o formato WAT e pode possuir expressões específicas d
a aplicação.
    Order: i32, // ordem que o advice deve executar.
    All (default: false): boolean, // indica se todas as funções são utilizadas
 na execução do *pointcut*, ou seja, para além das funções no código, também de
vem ser usadas as funções adicionadas pelo utilizador através da ferramenta.
    Smart(default: false): boolean, // indica se a transformação é inteligente.
  },
  Start: string, // código a ser adicionado à função inicial do módulo.
O código deve ter o formato WAT e pode possuir expressões específicas da aplica
ção.
  Templates: Mapa, // possui os templates que poderão ser
utilizados nos pointcuts.
}
```

Figura 124 - Estrutura YAML da linguagem de transformação do WasmManipulator

1. Sintaxe

Para facilitar a especificação da linguagem vão ser utilizados os seguintes tipos:

- *Object* - representa um objeto YAML, isto é, um elemento de chave-valor. Opcionalmente pode ter um tipo de valor específico, e para isso, é utilizada a sintaxe *Object<T>* (em que *T* é o tipo do valor).
- *Array* - representa um *array* YAML, isto é, uma lista. A lista deve ter um valor específico, e por isso, é sempre representada com a seguinte sintaxe *Array<T>* (em que *T* é o tipo do valor).
- *String* - representa um caracteres ASCII que formam um valor textual.
- *Identifier* - consiste numa *String* composta por caracteres alfanuméricos e o símbolo "_".
- *Type* - consiste numa *String* que representa os tipos de dados existentes na ferramenta.
- *Variable* - consiste numa *String* utilizada para declarar e inicializar variáveis.
- *Code* - representa o código no formato de uma *String*. Este é composto por código WAT e pode possuir determinadas instruções que se encontram especificadas abaixo.
- *CodeFunction* - é um subtipo de *Code*, utilizado apenas em funções.
- *CodeAdvice* - é um subtipo de *Code*, utilizado apenas em *advices*.
- *Pointcut* - consiste numa expressão *String* que representa a definição de um pointcut.
- *PointcutGlobal* - é um subtipo de *Pointcut*, com um formato específico para ser invocado por outros pointcuts.
- *PointcutAdvice* - é um subtipo de *Pointcut*, utilizado diretamente no *advice* e com a capacidade de recolher informação de contexto do módulo.
- *Template* - consiste numa *String* que servirá de template na busca de código.

Na Figura 125 encontra-se ilustrada a mesma estrutura especificada na Figura 124 mas com estes tipos aplicados aos campos.

```
{
  Pointcuts: PointcutGlobal,
  Aspects: {
    Context: {
      Variables: Map<Variable>,
      Functions: {
          Variables: Map<Variable>,
          Args: Array<{
            Name: Identifier,
            Type: Type,
          }>,
          Result: Type,
          Code: CodeFunction,
          Imported: {
            Module: String,
            Field: String,
          },
```

```
            Exported: String,
        },
      },
      Advices: {
        Pointcut: PointcutAdvice,
        Variables: Map<Variable>,
        Advice: CodeAdvice,
        Order: i32,
        All: boolean,
        Smart: boolean,
      },
    },
    Start: CodeFunction,
    Templates: Map<Template>,
}
```

Figura 125 – Estrutura YAML linguagem de transformação do WasmManipulator com os tipos especificados

1.1. String

Por definição, uma *string* é considerada um tipo de dados que consiste num array de bytes que armazena uma sequência de elementos usando um determinado tipo de codificação (Team T. , 2006). Contudo, no caso da ferramenta, *String* é uma derivação deste tipo de dados, cujos elementos são sempre caracteres, ou seja, este *array* é considerado sempre um elemento textual, usando ASCII como tipo de codificação.

1.2. Identifier

Identifier é um elemento textual do tipo *String*, no entanto, suporta apenas caracteres do tipo alfanumérico e o *underscore*. É utilizado para identificadores tais como nomes de funções e variáveis.

1.3. Type

O *Type* não é bem um tipo de dados, mas sim uma enumeração de *Strings* com os tipos de dados disponíveis para os elementos WASM na ferramenta. Estes tipos (Gohman, Lepesme, Qwerty2501, Spencer, & Um, 2021)[2] são os seguintes:

- *i32* - inteiro *32-bits*.
- *i64* - inteiro *64-bits*.
- *f32* - real *32-bits* (IEEE 754-2008).
- *f64* - real *64-bits* (IEEE 754-2008).
- *string* - possui as mesmas características do tipo *String*.
- *map*[*string*|*i32*|*f32*]*Type* - estrutura de dados do tipo mapa, ou seja, uma estrutura semelhante a uma tabela que permite indexar valores através de uma chave.
- []*Type* - estrutura de dados do tipo *array*, isto é, uma estrutura equivalente a uma lista de valores.

[2] Querty2501 é o nome de utilizador de um dos autores.

1.4. Variable

O tipo *Variable* consiste numa expressão do tipo *String* que permite declarar e inicializar uma dada variável. Para isso, esta variável deve sempre ser utilizada como valor num objeto YAML, sendo que a chave consistirá no nome da variável.

A sintaxe para uma variável é a seguinte: `@tipo < = @valor >?`.

Como o termo indica, o "tipo" consiste no tipo da variável a declarar. Este é do tipo *Type* e é obrigatório na expressão. A inicialização com "valor" é opcional, sendo que caso não seja incluído, a variável assume o valor nulo associado ao tipo. A informação relativa ao valor encontra-se descrita na Tabela 10.

Tabela 10 - Inicialização do valor das variáveis na ferramenta WasmManipulator

Tipo	***Valor***	***Nulo***	***Exemplo***
i32	i64	0	-1
i64	i64	0	1
f32	f32	0	1.1
f64	f64	0	-1.1
string	string	“”	“example”
map	array<[key,value]>	[]	[["key_1", 1],["key_2", 2]]
array	array<value>	[]	[1,2]

1.5. Code

A base para o código utilizado na linguagem da ferramenta é o WAT. Partindo desta base foram adicionadas as seguintes extensões exclusivas à linguagem da ferramenta:

- *Static Expressions* – são expressões interpretadas estaticamente, logo só têm acesso a contexto extático, como por exemplo, o nome de uma função (ver Secção 3.1).
- *Runtime Expressions* – são expressões sensíveis ao contexto e interpretadas em tempo de execução (ver Secção 3.2).
- *Runtime References* – são referências para variáveis interpretadas em tempo de execução(ver Secção 3.2.1).

As expressões (*static* e *runtime*) têm acesso ao contexto onde são aplicadas, sendo que este varia de acordo com o ambiente onde são utilizadas. O único contexto que é comum a todas as expressões é o contexto global, ou seja, funções e variáveis globais definidas no ficheiro de transformações. Os dados incluídos no contexto são acedidos através do respetivo identificador, por exemplo, se no ficheiro de transformações fosse declarada uma nova variável global com o nome “variable”, no interior das expressões, é necessário utilizar esse nome para substituir o identificador pelo índice da variável. Estas expressões são interpretadas pela ferramenta e numa fase final transformadas em código WAT.

1.5.1. CodeFunction

CodeFunction é um subtipo de *Code* que disponibiliza às expressões o acesso ao contexto da função criada. Desta forma, o utilizador consegue aceder aos argumentos da função, às suas variáveis locais, etc.

1.5.2. CodeAdvice

CodeAdvice é também um subtipo de *Code* que disponibiliza às expressões o acesso ao contexto do advice. Com isto, as expressões têm acesso não só aos dados definidos no *advice*, mas também à informação fornecida pelos *join-points* encontrados.

O código presente neste elemento consiste no código que irá substituir o conteúdo a que cada *join-point* está associado. Desta forma, segundo as linguagens orientadas a aspetos, este consiste numa operação "*around*". No entanto, com recurso à *keyword* `this` (Secção 3.2.2) que permite a inclusão do código associado, o utilizador é capaz de realizar as operações "*before*" e "*after*" sobre o *join-point* em questão.

1.6. Pointcut

Tal como na definição do *pointcut*, este tipo tem como objetivo encontrar um conjunto de *join*-points que coincidam com a expressão definida.

A sintaxe de um *Pointcut* varia de acordo com o tipo, contudo é sempre semelhante, parecendo-se com uma função *lambda* do JS:

```
(@parâmetro <, @parâmetro>*) => @expressão.
```

Os "parâmetros" variam de acordo com o tipo de *Pointcut*, no entanto a "expressão" mantém sempre o mesmo formato independentemente do tipo de *Pointcut Expression* (Secção 2).

1.6.1. PointcutGlobal

O *PointcutGlobal* é usado para definir um *pointcut* com propriedades globais, que pode ser incluído nos *pointcuts* associados a *advices*. Com isto, estes não possuem qualquer acesso ao contexto das funções, sendo que os parâmetros passados para o *lambda* são meras variáveis, desconhecidas pelo *pointcut*, e apenas controladas pelo invocador.

A sintaxe para o parâmetro do pointcut global é:

```
@tipo? @nome.
```

O "tipo" consiste no tipo da variável, é opcional, e é do tipo *Type*. Quando definido, é criada uma restrição sobre o tipo do parâmetro, quando este não é, o parâmetro pode assumir qualquer tipo. O "nome" é do tipo *Identifier* e consiste no nome da variável que será utilizado como referência na expressão do *Pointcut*.

1.6.2. PointcutAdvice

O *PointcutAdvice* é também usado para definir um *pointcut*, contudo disponibiliza o acesso aos dados de contexto das funções. Estes dados de contexto são passados como parâmetros, e podem ser utilizados tanto na expressão do *Pointcut* como no código do *advice*.

A expressão deste tipo de *Pointcut* pode invocar *Pointcuts* do tipo *PointcutGlobal*, passando-lhes as variáveis de contexto como argumentos.

A sintaxe para o parâmetro deste pointcut é:

```
<@tipo_variável.>?@tipo_contexto[@índice] @nome.
```

Para este tipo de *Pointcut* existem dois tipos de dados, o tipo da variável ("tipo_variável"), e o tipo de contexto ("tipo_contexto"). O tipo de variável é do tipo *Type* e refere-se à variável em si, o tipo de contexto é mais semelhante a um metadado, e refere-se ao tipo da variável no contexto de uma função. Este é composto por dois tipos: `param` (parâmetro) ou `local` (variável local). O "índice" pode assumir um valor numérico (ordem da variável dentro do seu contexto – semelhante ao espaço de índices, no entanto, há uma separação entre variáveis locais e parâmetros) ou o valor do índice em si (evitar uso em relação ao código original visto que no momento da transformação este pode ser imprevisível. O "nome" é do tipo *Identifier* e consiste no nome da variável, sendo este utilizado como referência na expressão do *Pointcut*.

1.7. Template

Por fim, o tipo *Template* consiste numa *String* que servirá de padrão na procura de código. Esta procura é feita com recurso à ferramenta Comby.

Para além de texto, o *Template* é composto por uma extensão parecida com as *static expressions*, no entanto, apesar de possuírem a mesma sintaxe, as expressões no *Template* são muito mais limitadas, tendo acesso apenas ao contexto presente no mesmo. Por esta razão, e por forma a distinguir ambos os tipos, estas vão ser chamadas de *template expressions* (Secção 3.3).

2. Pointcut Expressions

Pointcut expressions são um tipo expressões utilizadas na definição de um *Pointcut*, onde o utilizador combina um conjunto de funções *pointcut* através de operadores lógicos. Neste capítulo vão ser abordadas as várias funções disponibilizadas para criar uma destas expressões e quais os operadores disponíveis na ferramenta.

Os *pointcuts* disponíveis na ferramenta são os seguintes:

- `func` - encontra funções com uma determinada definição.
- `call` - encontra chamadas a funções que coincidam com uma determinada definição.
- `args` - encontra chamadas a funções que sejam chamadas com determinadas restrições nos argumentos.
- `returns` - encontra as instruções de retorno de um função.
- `template` - encontra um conjunto de instruções que coincidam com a definição do *template*.

Cada *pointcut* disponibiliza um conjunto de informação ao contexto do advice. Para aceder aos dados do *pointcut*, basta utilizar a *keyword* da função de *pointcut* dentro das expressões.

O acesso a estes dados deve ser feita de forma cautelosa, uma vez que, quando combinados com os operadores lógicos poderão ficar inconsistentes, uma vez que a expressão poderá provocar que determinados pointcuts tornem-se inválidos (por exemplo, na expressão `func || args` poderão existir situações em que apenas uma das duas funções de *pointcut* exista no *join-point* resultante).

Uma nota sobre os *join-points* encontrados: quando estes se sobrepõem, é sempre escolhido aquele que se encontra num nível de profundidade superior do código. O outro é ignorado, uma vez que se encontra inserido no primeiro. Por exemplo, na situação em que o *pointcut* `call` é executado sobre a expressão `(call $f0 (call $f1))`, apesar de coincidirem as instruções `(call $f1)` e `(call $f0 (call $f1)`, a que prevalece é a exterior (`(call $f0 (call $f1))`).

2.1. Pointcut func

O *pointcut* `func` filtra os *join-points* de acordo com a definição da função a que pertencem. Isto é, as instruções presentes no *join-point* devem pertencer a uma função que coincida com a configuração definida pelo utilizador.

Caso a execução do *pointcut* seja feita sobre um ambiente vazio (primeira operação a ser executada), este cria um *join-point* para cada função que coincide com a definição, envolvendo todas as instruções presentes nessa função.

2.1.1. Sintaxe

A sintaxe para o *pointcut* `func` é:

```
func(@retorno @função(@parâmetros?)<, @scope>?).
```

Os elementos da sintaxe poderão assumir múltiplos valores. Na Tabela 11 estão descritos estes elementos e a respetiva sintaxe.

Tabela 11 - Elementos da função de *pointcut* func

Elemento	***Descrição***		
retorno	Tipo de retorno.		
	Sintaxe	***Significado***	***Exemplo***
	*	qualquer tipo de retorno	*
	void	sem retorno	void
	@tipo	tipo do retorno	i32
	%@ident%	designação do tipo fica armazenado numa variável; qualquer tipo de retorno	%var%
	%@ident:void%	designação do tipo fica armazenado numa variável; sem retorno	%var:void%
	%@ident:@tipo%	designação do tipo fica armazenado numa variável; tipo do retorno	%var:i32%

<table>
<tr><th>Elemento</th><th>Descrição</th></tr>
<tr><td></td><td>Observações:
"tipo" é do tipo Type
"ident" é do tipo Identifier</td></tr>
<tr><td>função</td><td>Identificador da função.
<table>
<tr><th>Sintaxe</th><th>Significado</th><th>Exemplo</th></tr>
<tr><td>*</td><td>qualquer identificador</td><td>*</td></tr>
<tr><td>@ nome</td><td>nome exportado da função</td><td>fn_name</td></tr>
<tr><td>/@regex/</td><td>expressão regular para o nome exportado da função</td><td>/\w+/</td></tr>
<tr><td>$@índice_nome</td><td>índice textual da função</td><td>$f1</td></tr>
<tr><td>[@índice_ordem]</td><td>índice de ordem da função</td><td>[1]</td></tr>
<tr><td>%@ident%</td><td></td><td>%fn%</td></tr>
<tr><td>%@ident:@nome%</td><td>nome exportado fica armazenado numa variável; nome exportado da função</td><td>%fn:fn_name%</td></tr>
<tr><td>%@ident:/@regex/%</td><td>nome exportado fica armazenado numa variável; expressão regular para o nome exportado da função</td><td>%fn:/\w+/%</td></tr>
<tr><td>%@ident:$@índice_nome%</td><td>índice (textual) fica armazenado numa variável; índice textual da função</td><td>%fn:$f1%</td></tr>
<tr><td>%@ident:[@índice_ordem]%</td><td>índice (ordem) fica armazenado numa variável; índice de ordem da função</td><td>%fn:[1]%</td></tr>
</table>
Observações:
"nome", "ident", "índice_nome" são do tipo Identifier
"regex" é do tipo String
"índice_ordem" é do tipo i32 (Type)</td></tr>
<tr><td>parâmetros</td><td>Parâmetros da função.
<table>
<tr><th>Sintaxe</th><th>Significado</th><th>Exemplo</th></tr>
<tr><td></td><td>sem parâmetros</td><td></td></tr>
<tr><td>..</td><td>qualquer configuração para os parâmetros</td><td>..</td></tr>
<tr><td><i>@parâmetro</i> <, <i>@parâmetro</i>>*</td><td>tipo do retorno</td><td>i32 %p0%, i64</td></tr>
</table>
Observações:
A sintaxe para o "parâmetro" está definida abaixo</td></tr>
<tr><td>parâmetro</td><td>Parâmetro da função.
<table>
<tr><th>Sintaxe</th><th>Significado</th><th>Exemplo</th></tr>
<tr><td>*</td><td>qualquer parâmetro na respetiva ordem</td><td>*</td></tr>
</table></td></tr>
</table>

Elemento	*Descrição*		
	@tipo	parâmetro de um tipo específico	i32
	**** %@ident%***	parâmetro de qualquer tipo armazenado numa variável	* %p0%
	@tipo %@ident%	parâmetro de um tipo específico armazenado numa variável	i32 %p0%
	Observações: • "tipo" é do tipo *Type* • "ident" é do tipo *Identifier*		
scope	*Scope* da função no módulo.		
	Sintaxe	***Significado***	***Exemplo***
		a função pode conter qualquer *scope*	
	imported	função importada	imported
	exported	função exportada	exported
	internal	função interna (privada, ou seja, nem importada, nem exportada)	internal

2.1.2. Dados de Contexto

Na Tabela 12 está representado o modelo de dados (`Func`) que representa a informação que é acrescentada ao contexto do *advice*. Estes dados estão associados à função que possui as instruções incluídas no *join-point*. Os dados estão contidos no identificador `func`, que poderá ser invocado nas expressões do código.

Tabela 12 – Modelo de dados de contexto da função de *pointcut* func

Nome	*Tipo*	*Descrição*
Index	string	Nome do índice da função.
Order	i32	Ordem do índice da função.
Name	string	Caso exportada, consiste no nome exportado da função. Caso contrário, é igual ao nome do índice.
Params	Array<string>	Lista com os nomes dos índices dos parâmetros.
ParamTypes	Array<string>	Lista com os tipos dos parâmetros.
TotalParams	i32	Número total dos parâmetros.
Locals	Array<string>	Lista com os nomes dos índices das variáveis locais.
LocalTypes	Array<string>	Lista com os tipos das variáveis locais.
TotalLocals	i32	Número total das variáveis locais.
ResultType	string	Tipo do resultado da função.
Code	string	Instruções da função em formato textual.
IsImported	boolean	Se a função é importada.

IsExported	boolean	Se a função é exportada.
IsStart	boolean	Se a função é executada inicialmente.

2.2. Pointcut call

O *pointcut* `call` tem como objetivo encontrar instruções que correspondam a chamadas a funções com uma determinada configuração. Os *join-points* gerados pelo *pointcut* correspondem à instrução na globalidade, incluindo não só a instrução da chamada, mas também as instruções correspondentes aos argumentos passados à função.

2.2.1. Sintaxe

A sintaxe utilizada para a configuração é igual à sintaxe para o *pointcut* `func`. Isto deve-se ao facto de que ambos dependem da configuração da função para operar.

Com isto, a sintaxe para o *pointcut* `call` é:

```
call(@retorno @função(@parâmetros?)).
```

A descrição dos vários elementos da sintaxe encontra-se expressa na Tabela 11, da secção com o *pointcut* `func` (Secção 2.1).

2.2.2. Dados de Contexto

À semelhança do *pointcut* `func`, o *pointcut* `call` acrescenta dados relacionados com a função a que o *join-point* está associado. Contudo, estes dados tanto existem para a função que fez a chamada, como para a função que foi invocada, e por isso são encapsulados campos diferentes. Para além desses campos, são também incluídos os dados relacionados com os argumentos passados na instrução da chamada. Este modelo de dados encontra-se representado na Tabela 13. Os dados estão contidos no identificador `call`, que poderá ser invocado nas expressões do código.

Tabela 13 – Modelo de dados de contexto da função de *pointcut* call

Nome	***Tipo***	***Descrição***
Callee	Func (Tabela 12)	Dados da função invocada.
Caller	Func (Tabela 12)	Dados da função que invocou.
Args	Array<Arg> (Tabela 14)	Lista com a informação dos argumentos.
TotalArgs	i32	Número total de argumentos.

O objeto `Arg` possui a informação relativa ao argumento de uma função. O seu modelo de dados encontra-se representado na Tabela 14.

Tabela 14 - Modelo de dados para os dados de contexto do tipo “argumento"

Nome	*Tipo*	*Descrição*
Type	string	Tipo do argumento.
Order	i32	Ordem do argumento na chamada.
Instr	string	Código WAT do argumento.

2.3. Pointcut args

O *pointcut* `args`, tal como o *pointcut* `call`, tem como objetivo encontrar chamadas a funções, no entanto, a pesquisa para este é feita com recurso às variáveis de contexto passadas como parâmetros ao *Pointcut*.

Ao aceitar apenas variáveis de contexto para a pesquisa faz com que os resultados a obter sejam muito específicos, uma vez que a instrução da chamada deve ter obrigatoriamente nos seus argumentos o acesso a essas variáveis (instrução `local.get`).

2.3.1. Sintaxe

A sintaxe para o *pointcut* `args` é:

```
args(<@argumento <, @argumento>*>?).
```

O *pointcut* aceita qualquer número de argumentos, sendo que cada "argumento" é do tipo *Identifier* e corresponde a uma variável de contexto do *Pointcut*.

2.3.2. Dados de Contexto

O modelo de dados do *pointcut* `args` é igual ao do *pointcut* `call`, e por isso encontra-se representado na Tabela 13 da respetiva secção do *pointcut*. Este é também composto pelos dados referentes a ambas as funções (a função invocada e a que invocou a chamada) e pelos dados referentes aos argumentos passados na instrução. Os dados estão contidos no identificador `args`, que poderá ser invocado nas expressões do código.

2.4. *Pointcut* returns

O *pointcut* `returns` tem como objetivo encontrar todas as instruções de retorno de uma dada função. Este aceita na sua configuração um dado tipo, que permite filtrar os *join-points* por tipo de retorno.

2.4.1. Sintaxe

A sintaxe para o *pointcut* `returns` é:

```
returns(@tipo).
```

O "tipo" consiste no tipo de dados esperado no retorno, sendo que também é aceite o valor `*` para indicar que os *join-points* não requerem nenhum tipo de retorno em específico.

2.4.2. Dados de Contexto

O modelo de dados correspondente aos dados de contexto adicionados após a execução do *pointcut* `returns` está representado na Tabela 15. Estes dados estão relacionados à instrução

de retorno a que o *join-point* está associado. Os dados estão contidos no identificador `returns`, que poderá ser invocado nas expressões do código.

Tabela 15 - Modelo de dados de contexto da função de *pointcut* returns

Nome	***Tipo***	***Descrição***
Func	Func (Tabela 12)	Dados da função que contém a instrução de retorno.
Type	string	Tipo da instrução de retorno.
Instr	string	Código WAT do instrução de retorno.

2.5. *Pointcut* template

Este *pointcut* é utilizado para realizar a pesquisa por padrão na ferramenta. Para isso, deve ser referenciado o respetivo *template* que servirá de padrão durante a pesquisa por *join-points*.

2.5.1. Sintaxe

A sintaxe para o *pointcut* `template` é:

```
template(<@template <, @validação>).
```

O "template" indicado no *pointcut* corresponde a uma das chaves presente no ficheiro de transformação, dentro do objeto *Templates*, que está associada ao *template* que servirá de padrão na pesquisa.

A "validação" é do tipo *boolean* (`true` ou `false`), e serve para indicar se o *template* está a ser executado apenas como forma de validação ou não. Por predefinição esta configuração está desativada, o que significa que os resultados obtidos apenas contém as instruções que coincidem diretamente com a definição do *template*. Ao ativar a configuração, o *template* servirá apenas como um padrão de validação, onde não é feita a filtragem das instruções, e por isso, qualquer entrada que possua no seu conteúdo o padrão definido no *template* é adicionado aos resultados. Desta forma, caso um *join-point* seja válido para um dado *template*, todas as instruções deste *join-point* se mantêm.

2.5.2. Dados de Contexto

Ao contrário dos outros *pointcuts*, o identificador adicionado ao contexto do *advice* corresponde à chave do *template* incluído na definição, e não o nome da própria função de *pointcut*. Com isto, são extraídos as várias variáveis definidas no *template* e encapsulados no identificador de contexto (chave do *template*). Depois, o seu acesso e manipulação é realizado através das funções disponibilizadas nas *static expressions*.

7.3.1.1. Operadores Lógicos

Estes *pointcuts* são combinados com recurso aos seguintes operadores lógicos:

- `&&` - corresponde ao operador lógico "*And*".
- `||` - corresponde ao operador lógico "*Or*".
- `()` - usado no agrupamento de operações.

3. Code Expressions

3.1. Static Expressions

As *static expressions*, ou expressões estáticas, permitem a manipulação do código, o acesso aos dados do contexto, e a realização de operações sobre informação conhecida em *compile time* (informação estática ou proveniente do contexto do *advice*).

Este tipo de expressões representam o principal sistema utilizado pela ferramenta para implementar no código um paradigma orientado a aspetos. Isto deve-se ao facto de não só permitirem manipular conteúdo estático, como também as instruções presentes no *join-point*. Estas instruções estão disponíveis através do identificador `this`. Como resultado existe uma forma flexível de interagir com cada *join-point*, onde é possível reproduzir as operações comuns às linguagens AOP, tais como, inserir "antes" ou "depois", "substituir" as instruções, etc. Para além disso, a ferramenta também permite a transformação destes dados através das funções de transformação (Secção 3.1.4).

3.1.1. Sintaxe

A sintaxe presente nas *static expressions* é a seguinte:

```
%@variável<:@método>*%.
```

A "variável" refere-se ao identificador da variável existente no contexto do *advice* ou da função onde é incluída. Relativamente ao "método", este consiste numa função de transformação, em que a sua utilização segue um paradigma funcional (Noleto, 2020), ou seja, são encadeados de forma imperativa, formando uma sequência de operações que ao receber o mesmo valor, devolvem sempre o mesmo resultado. Cada "método" possui a sua própria sintaxe detalhada na Secção 3.1.4.

3.1.2. Contexto

As *static expressions* podem ser utilizadas tanto em funções como em *advices*. Desta forma, o contexto depende do sítio onde é aplicada a expressão.

Os dados presentes no contexto disponibilizado para as expressões incluídas na definição de funções são os seguintes:

- Parâmetros da função.
- Variáveis locais.
- Variáveis globais.
- Funções declaradas no ficheiro de transformação.

Relativamente, ao contexto disponibilizado nas expressões incluídas no código de um *advice*, este é composto pelos seguintes dados:

- O código do *join-point* (identificador `this`).
- Variáveis disponibilizadas pelos *pointcuts*.
- Parâmetros do *pointcut*.
- Variáveis locais definidas no *advice*.

- Variáveis globais.
- Funções declaradas no ficheiro de transformação.

3.1.3. Tipos de Variáveis

Nas *static expressions* os dados possuem tipos distintos. Cada um deste tipo possui um conjunto de funções de transformação associado, que por sua vez, poderá ter comportamentos também diferentes. Desta forma, foram criados os seguintes tipos de dados:

- *string* - equivalente ao tipo de dados *String*.
- *string_slice* - consiste num *array* de dados com o tipo *String*.
- *template_search* - corresponde ao resultado obtido num dado *template*.
- *object* - consiste num objeto composto. Pode assumir o tipo *array*, *mapa*, *objeto*, *string*, *i32*, *i64*, *f32*, *f64* ou *null*.

Quando estas expressões são convertidas para WAT, o seu valor é automaticamente convertido para o respetivo valor do tipo *string*. Neste caso, é invocada a função de transformação `string()` (Secção 3.1.4) sobre o resultado da expressão.

3.1.4. Funções de Transformação

Cada função de transformação recebe um valor de entrada e devolve o respetivo resultado de acordo com a operação executada. O tipo dos dados de entrada/saída varia de acordo com a função aplicada. Para além disto, a configuração dos parâmetros da função também varia com o tipo de função.

Na Tabela 16 encontram-se representadas todas as funções de transformação disponíveis na ferramenta. Para cada função é apresentada uma pequena descrição, a sua sintaxe, exemplos de utilização e os tipos dos valores de entrada e saída.

Tabela 16 - Funções de transformação para *static expressions*

Função	***Descrição***
string	Converte o valor de entrada numa *String*.
	Sintaxe**:** string().
	Exemplos: 1. ["1","2","3"]:string() → "123". 2. object<{k1:"v1"}>:string() → "{\"k1\":\"v1\"}".
	Tipos: • string → string . • string_slice → string. • template_search → string. • object → string.
type	Devolve o tipo do valor do valor de entrada.
	Sintaxe: type().

Função	*Descrição*
	Exemplos: 1. ["1","2","3"]:type() → "string_slice".
	Tipos: • string → string . • string_slice → string. • template_search → string. • object → string.
order	Devolve a ordem do índice associado a uma dada função.
	***Sintaxe*:** order().
	Exemplos: 1. "$f1":string() → "1".
	Tipos: • string → string . • string_slice → string. • template_search → string. • object → string.
map	Cria um novo valor a partir do valor de entrada, chamando uma função específica em cada elemento presente no respetivo valor de entrada.
	***Sintaxe*:** map((@entrada <, @índice>?) => @expressão). A "entrada" consiste no identificador que referencia cada elemento presente no valor de entrada. O "índice" é opcional e corresponde ao índice numérico da iteração. A "expressão" consiste na expressão que será interpretada e originará uma entrada no valor de saída (no lugar do elemento de entrada). Esta "expressão" será sempre convertida para *string*.
	Exemplos: 1. ["1","2","3"]:map((v) => "num " + v) → ["num 1","num 2","num 3"]. 2. object<{k1:"v1",k2:"v2"}>:map((v) => v) → ["v1","v2"]. ***3.*** object<["v1","v2"]>:map((v) => v) → ["v1","v2"].
	Tipos: • string → string. • string_slice → string_slice. • template_search → string_slice. • object → string_slice.
repeat	Repete o valor de entrada um dado número de vezes.
	***Sintaxe*:** repeat(@n). "n" é um valor numérico referente ao número de vezes que o valor de entrada vai ser repetido. Uma particularidade da função é que quando a entrada é um *array*, a saída não será um *array* de *arrays*, mas sim um *array* com cada valor repetido "n" vezes.

Função	*Descrição*
	Exemplos: 1. "123":repeat(2) → ["123","123"]. 2. ["1","2","3"]:repeat(2) → ["1","1","2","2","3","3"]. 3. object<["1","2","3"]>:repeat(2) → ["1","1","2","2","3","3"]. ***4.*** object<{k1:"1"}>:repeat(2) → ["{\"k1\":\"1\"}","{\"k1\":\"1\"}].
	Tipos: • string → string_slice. • string_slice → string_slice. • template_search → string_slice. • object → string_slice.
join	Conecta os elementos do valor de entrada utilizando um determinado separador.
	Sintaxe**:** join(@separador). O "separador" é utilizado para juntar os vários elementos para um resultado do tipo *string*. Este é sempre convertido para *string*.
	Exemplos: 1. ["1","2"]:join(",") → "1,2". ***2.*** object<{k1:"1",k2:"2"}>:join(",") → "1,2".
	Tipos: • string → string. • string_slice → string. • object → string.
split	Divide a entrada numa *array* de *strings*.
	Sintaxe**:** split(@separador). O "separador" é utilizado para separar os vários elementos para um resultado do tipo *string*. Este é sempre convertido para *string*.
	Exemplos: 1. "123":split("") → ["1","2","3"]. 2. "12345":split("2") → ["1","345"].
	Tipos: • string → string_slice.
count	Devolve o número de elementos do valor de entrada.
	Sintaxe**:** count().
	Exemplos: 1. "321":count()→ "3". 2. ["4","3","2","1"]:count() → "4". 3. object<{k1:"1",k2:"2"}>:count() → "2".

Função	***Descrição***
	Tipos: • string → string . • string_slice → string. • template_search → string. • object → string.
contains	Devolve se o valor de entrada contém um dado valor/chave.
	***Sintaxe*:** contains(@valor). O "valor" é sempre convertido para *string*.
	Exemplos: 1. "321":contains("2") → "true". 2. ["4","3","2","1"]:contains("5") → "false". ***3.*** object<{k1:"1",k2:"2"}>:contains("k2") → "true".
	Tipos: • string → string . • string_slice → string. • template_search → string. • object → string.
assert	Interrompe a cadeia de operações caso a condição não seja atendida.
	***Sintaxe*:** assert((@entrada => @condição). A "entrada" consiste no identificador que referencia o valor de entrada. A "condição" será sempre transformada sempre num valor booleano.
	Exemplos: 1. "123":assert((v) => v - 1 == 122) → "123". 2. "123":assert((v) => v - 1 != 122) → "".
	Tipos: • string → string | "". • string_slice → string_slice | "". • template_search → template_search | "". • object → object | "".
replace	Substitui conteúdo do valor de entrada, ou parte do mesmo, por um novo valor.
	***Sintaxe*:** replace(@valor_anterior, @valor_novo). O "valor_anterior" consiste no valor que deve ser substituído pelo "valor_novo". Ambos os parâmetros, "valor_anterior" e "valor_novo", serão sempre convertidos para *string*.

Função	*Descrição*
	Exemplos: 1. "123":replace("2","5") → "153". 2. ["1","2","3"]:replace("1","5") → ["5","2","3"]. 3. object<{k1:"1",k2:"2"}>:replace("k2","3") → object<{k1:"1",k2:"3"}>. 4. search<{result:"1|2",values:{k1:"1",k2:"2"}}>:remove("k1","3") → search<{result:"3|2",values:{k1:"3",k2:"2"}}>.
	Tipos: • string → string. • string_slice → string_slice. • template_search → template_search. • object → object.
remove	Remove parte do conteúdo do valor de entrada.
	***Sintaxe*:** remove(@valor). O "valor" corresponde à configuração que será removida do valor de entrada. Este é sempre convertido para *string*.
	Exemplos: 1. "123":remove("23") → "1". 2. ["1","2","3"]:remove("2") → ["1","3"]. 3. object<{k1:"1",k2:"2"}>:remove("k2") → object<{k1:"1"}>. ***4.*** search<{result:"1|2",values:{k1:"1",k2:"2"}}>:remove("k1") → search<{result:"|2",values:{k2:"2"}}>.
	Tipos: • string → string. • string_slice → string_slice. • template_search → template_search. • object → object.
filter	Filtra elementos do valor de entrada segundo uma determinada condição.
	***Sintaxe*:** filter((@entrada <, @índice>?) => @expressão). A "entrada" corresponde ao elemento do valor de entrada. O "índice" é opcional e corresponde ao índice numérico da iteração. A "expressão" consiste na expressão que será interpretada e dependendo do resultado, o valor será adicionado (ou não) no valor de saída.
	Exemplos: 1. "147":filter((v)=>v%2!=0) → "17". 2. ["1","4","7"]:filter((v)=>v%2==0) → ["4"]. 3. object<{k1:"1",k2:"2"}>:filter((v)=>v%2==0) → ["2"]. ***4.*** search<{result:"1|2",values:{k1:"1",k2:"2"}}>:filter((v)=>v!="2") → "1|".

Função	*Descrição*
	Tipos: • string → string. • string_slice → string_slice. • template_search → string. • object → string_slice.
slice	Altera o conteúdo de um valor de entrada selecionando o intervalo indicado.
	***Sintaxe*:** slice(@início <, @fim>?). O "início" e o "fim" correspondem aos índices do intervalo que corresponderá ao valor de saída. Estes valores devem ser numéricos, sendo que o "fim" é opcional, assumindo o tamanho do valor de entrada por predefinição.
	Exemplos: 1. "147":slice(1) → "47". 2. ["1","4","7"]:slice(0,1) → ["1"]. 3. object<{k1:"1",k2:"2"}>:slice(1) → ["2"].
	Tipos: • string → string. • string_slice → string_slice. • template_search → string. • object → string_slice.
splice	Altera o conteúdo de um valor de entrada removendo o intervalo indicado.
	***Sintaxe*:** splice(@início <, @fim>?). O "início" e o "fim" correspondem aos índices do intervalo a remover. Estes valores devem ser numéricos, sendo que o "fim" é opcional, assumindo o tamanho do valor de entrada por predefinição.
	Exemplos: 1. "147":splice(1) → "1". 2. ["1","4","7"]:splice(0,1) → ["4","7"]. ***3.*** object<{k1:"1",k2:"2"}>:splice(1) → ["1"].
	Tipos: • string → string. • string_slice → string_slice. • template_search → string. • object → string_slice.
select	Seleciona um sub-valor do template.
	***Sintaxe*:** select(@ident) . O "ident" corresponde ao identificador do valor no resultado do *template*. Este é sempre convertido para *string*.

Função	*Descrição*
	Exemplos: ***1.*** search<{result:"1|2",values:{k1:"1",k2:"2"}}>:select("k1") → search<{result:"1",values:{}}> .
	Tipos: • template_search → template_search.
reverse	Reverte a ordem dos elementos de entrada.
	***Sintaxe*:** reverse().
	Exemplos: 1. ["1","2","3"]:reverse() → ["3","2","1"]. 2. "123":reverse() → "321".
	Tipos: • string → string. • string_slice → string_slice. • template_search → string. • object → object.

3.1.5. Palavras Reservadas (*Keywords*)

As *keywords* reservadas para as *static expressions* são as seguintes:

- `this` que contém as instruções associadas a um *join-point*.
- `func`, `call`, `args` e `returns` que representam os dados de contexto fornecidos pelos vários *pointcuts*.
- o caractere `;` permite a junção de múltiplas *static expressions* numa só.

Nota: A ferramenta possui algumas palavras reservadas, ou *keywords*, que têm uma finalidade específica e por isso não podem, ou não devem ser usadas como identificadores de variáveis e funções. A utilização das *keywords* variam de acordo com o tipo de expressão onde são utilizadas.

3.1.6. Observações

Além da informação exposta neste capítulo, existem as seguintes observações a fazer relativamente às *static expressions*:

- Para além dos identificadores de contexto, é possível iniciar uma sequência de operações sobre um valor estático do tipo *string* (`%""%`).
- Suportam modificadores numéricos (cálculos) dentro dos *lambda*.
- As *keywords* para estas expressões não são as mesmas que para as *runtime expressions*.
- O resultado final é sempre convertido para *string*.
- Os comentários inline (iniciados através de `;;`) devem possuir uma linha vazia extra para que sejam interpretados como tal. Caso contrário, o código escrito nessa linha

é ignorado. Desta forma, o aconselhável é a utilização de blocos de comentários (entre `(;` e `;)`).

3.2. Runtime Expressions

O objetivo principal destas expressões consiste na geração de código em *runtime*, ou seja, o código das transformações é sensível ao contexto em que se encontra a executar. Para além disto, também serão utilizadas para permitir a interação com valores cujo tipo é desconhecido pelo WASM (*strings*, mapas e *arrays*).

As *runtime expressions* estão fortemente acopladas com o JS (decisão tomada com base na explicação da Secção 2.5), uma vez que todo o processo de execução vai ser realizado com recurso à função JS `eval` (MDN Contributors, eval(), 2021).

Com a utilização destas expressões não só será possível a utilização de novos tipos, conseguindo assim a implementação de algumas funcionalidades que utilizando WASM "puro" seriam impossíveis (ou quase impossíveis), tais como, *logging* (Secção 6.2.6.1) ou *caching* (Secção 6.2.6.2), como também a capacidade de executar expressões complexas com os dados do contexto em tempo de execução.

Apesar da utilização deste tipo de instruções possuir diversos benefícios, também existem algumas desvantagens que podem ser limitadoras para o utilizador. Uma das desvantagens é o facto do utilizador necessitar não só do código WASM gerado, mas também do código JS, sendo que a interação com o programa WASM deve ser feita com recurso a este último ficheiro, e não como o módulo WASM. Outra desvantagem é que o tamanho do ficheiro WASM aumenta consideravelmente de tamanho devido à geração de instruções necessárias à comunicação com o cliente. Por último, pode existir uma diminuição de desempenho no programa, uma vez que o programa não se resume apenas à utilização de instruções WASM.

A utilização de *runtime expressions* possui algumas restrições. Estas restrições estão relacionadas com a instrução onde é invocada. Quando é invocada na "raiz" da função, assume o tipo do retorno da função. Se for invocada dentro das instruções `call`, `local.set/tee` e `global.set`, dependem do tipo do primeiro argumento da instrução, quer seja função no caso da `call`, quer seja uma variável no caso do `local.set/tee` e `global.set`. Por último, dá para incluir estas expressões dentro de instruções WASM onde é possível conhecer o tipo do valor esperado para o respetivo argumento onde a expressão é aplicada (por exemplo, para a instrução `i32.add` é possível obter os tipos de ambos os parâmetros - *i32*). Todas as restantes instruções não permitem a utilização deste tipo de expressões.

Com isto, é possível definir a seguinte sintaxe para as diversas formas de aplicação destas expressões:

- *@índice* = valor do índice
- *@expressão* = código JS + referências
- *@referência* = nome da variável
- *@runtime_expression* = /*@expressão*/
- *@runtime_reference* = #*@referência*

- *@var_ident* = *@índice* | *@runtime_reference*
- *@call* = (call *@índice* *@runtime_expression*)
- *@set_local* = (local.set *@var_ident* *@runtime_expression*)
- *@tee_local* = (local.tee *@var_ident* *@runtime_expression*)
- *@set_global* = (global.set *@var_ident* *@runtime_expression*)
- As restantes instruções disponíveis não possuem qualquer formato predefinido, sendo que deve ser utilizada a sintaxe "*runtime_expression*" no lugar das expressões.

3.2.1. Runtime References

As *runtime references* têm como objetivo referenciar uma dada variável no código para as operações *runtime*. Estas tanto podem ser usadas para identificar variáveis dentro das *runtime expressions*, como também para referenciar variáveis que serão alteradas através das instruções `local.set/tee` e `global.set` ou retornos de uma função.

No que toca ao primeiro caso, as referências vão permitir que a ferramenta consiga identificar quais as variáveis que devem ser substituídas pelo respetivo valor em tempo de compilação, e assim, proceder às respetivas alterações de código. No segundo caso, estas referências devem sempre ser combinadas com *runtime expressions*, uma vez que não só a interpretação dessas expressões é responsável por atribuir a referência correta à *runtime reference*, como também, as variáveis declaradas neste caso não são inspecionadas em tempo de compilação, e desta forma, não serão detetáveis no momento da execução. Como consequência, o valor pode não existir ou encontrar-se num estado obsoleto no momento em que a referência é executada (Figura 126). Para contornar o problema, é aconselhado que quando se necessita de uma variável neste tipo de referência, antes exista uma instrução que a utilize numa *runtime expression* (ver Secção 3.2). A utilização das referências só é obrigatória quando é feito o acesso a membros de variáveis do tipo mapa ou *array* (por exemplo, `array[1]` ou `mapa["chave"]`).

```
(local.set #index /#index/) ;; A utilização duma instrução vai registar a variá
vel index em tempo de compilação.
(local.set #mapa[index] /#valor/) ;; e assim será possível usar na runtime refe
rence.
(...)
(local.set #mapa[index] /#valor/) ;; Valor para a variável index pode encontrar
-se obsoleto.
(...)
(local.set #mapa[#index] /#valor/) ;; Erro! A referência #index está a ser util
izada com índice de outra runtime reference sem ser dentro de uma runtime expre
ssion.
```

Figura 126 - Código com limitação das *runtime references*

3.2.2. Palavras Reservadas (*Keywords*)

As *keywords* definidas para as *runtime expressions* são todas as *keywords* existentes no JS, e para além disto, a *keyword* `return_`, que representa internamente o valor de retorno de uma função com o tipo complexo.

Nota: A ferramenta possui algumas palavras reservadas, ou *keywords*, que têm uma finalidade específica e por isso não podem, ou não devem ser usadas como identificadores de variáveis e funções. A utilização das *keywords* variam de acordo com o tipo de expressão onde são utilizadas.

3.3. Template Expressions

As *template expressions* são utilizadas para definir o código dos *templates* que poderão ser utilizados na expressão do *pointcut*, para a realização de uma pesquisa por padrão. Durante o processo de pesquisa, as variáveis que vão sendo recolhidas são incluídas no contexto do *advice*, encapsuladas no identificador referente à chave (nome) do *template*. No final da pesquisa, este identificador é convertido num modelo interno do tipo `search_template` (Secção 3.1.3), onde poderá ser acedido e manipulado com recurso às *static expressions* aplicadas no código de transformação da ferramenta.

Os *templates* estão definidos no objeto *Template* do ficheiro de transformações, e são identificados através do valor da chave onde estão inseridos, isto é, o seu nome.

3.3.1. Template Keywords

Estas expressões são compostas por uma linguagem específica que combina o WAT com uma sintaxe semelhante às *static expressions*, as *template keywords*, mas cujo objetivo é muito distinto. Enquanto que as *static expressions* são interpretadas e convertidas para WAT, as *template keywords* servem como um *placeholder* no padrão que poderá ficar associado a uma dada variável.

As *template keywords* são capazes de combinar *templates* uns com os outros. Para isso, estas suportam a utilização de funções, que possuem uma sintaxe semelhante às funções de transformação das *static expressions*, e permitem que uma dada variável respeite uma dada restrição segundo outro *template*. Estas restrições não só englobam que as variáveis coincidam (ou não) com o padrão definido no *template* integrado, como também, declaram variáveis que devem ser definidas nesse *template*.

Na Tabela 17 encontram-se representadas as várias funções disponíveis na ferramenta. Para cada função é feita uma breve descrição e apresentada a devida sintaxe. Na sintaxe, o “template" corresponde ao nome do template a integrar, e o "var_ident" corresponde ao identificador que deve estar definido no template a integrar.

Tabela 17 - Funções para combinação de *templates* nas *template keywords*

Função	***Descrição***	***Sintaxe***
include	O valor do identificador deve coincidir com o template indicado.	include(@template)
include_one	O valor do identificador deve coincidir com pelo menos um dos templates indicados.	include_one(@template <, @template>*)
include_all	O valor do identificador deve coincidir com todos os templates indicados.	include_all(@template <, @template>*)
not_include	O valor do identificador não pode coincidir com o template indicado.	not_include(@template)
not_include_one	O valor do identificador não pode coincidir com nenhum dos templates indicados.	not_include_one(@template <, @template>*)
not_include_all	O valor do identificador não pode coincidir com todos os templates indicados. Ou seja, só não é válido se coincidir com todos os templates.	not_include_all(@template <, @template>*)
define	O *template* a integrar deve obrigatoriamente definir os identificadores indicados. Desta forma, a função *define* só é permitida quando é procedida pelas funções *include*, *include_one* e *include_all*.	define(@var_ident)

3.3.2. Funcionamento

A aplicação dos *templates* é feita com recurso ao Comby (Comby, 2021). Na preparação da *query* as *template keywords* são sempre substituídas por um "*Named Match*", permitindo assim que a ferramenta associe uma dada *keyword* à variável correspondente. Os resultados obtidos são interpretados pela ferramenta e armazenados numa estrutura central recursiva, existindo a possibilidade de serem executados mais do que um *template* de acordo com a definição do utilizador. Esta estrutura contém a respetiva iteração com o valor encontrado e os valores das variáveis que compõem essa iteração. A ferramenta utiliza apenas as primeiras iterações encontradas, ou seja, caso sejam encontradas várias correspondências para a mesma *query*, apenas a primeira será utilizada. Esta limitação foi estabelecida com o intuito de simplificar a utilização dos *templates* para o utilizador, uma vez que, após a transformação do código associado à primeira iteração, o código associado às restantes iterações encontrar-se-ia desatualizado, e como consequência, iria ocorrer um erro de execução, ou no pior cenário, o resultado obtido com as transformações seria enganador ou sem sentido para o utilizador. Contudo, é fornecida uma forma de contornar esta limitação que passa pela utilização de vários *advices* com a mesma definição. O único desafio desta abordagem seria conhecer o número de *advices* que é necessário executar, no entanto, o

utilizador pode sempre executar a ferramenta até que não existam novas alterações, e assim, garantir que todas as iterações são devidamente transformadas.

4. Modo Inteligente (Smart)

Este modo inteligente é configurado para cada um dos *advices* declarados no ficheiro de transformações, e define o modo como as transformações irão operar. Caso este modo esteja ativo, a transformação tem em conta o valor de retorno das instruções referentes ao *join-point* em questão, e procede às transformações extras que permitem manter o mesmo valor de retorno.

Neste modo o utilizador pode definir uma instrução `target` no código do *advice* que será a instrução que servirá de retorno para o código que está a ser modificado. Caso não seja definido nenhum `target`, a ferramenta faz uma pesquisa pela instrução que anteriormente existia. Se for encontrada, a ferramenta assume a instrução como sendo o `target`, mas se essa instrução não existir no novo código, não é realizada qualquer transformação inteligente.

Para perceber melhor o conceito, a seguir será apresentado um exemplo conceptual. Neste exemplo, as instruções que se encontram a ser modificadas são ambas as chamadas existentes na instrução de adição. Esta modificação está relacionada com a instrumentação de código, sendo que deve ser adicionada uma função antes e depois de qualquer chamada realizada no código. A Figura 127 representa o código WAT original, a Figura 128 representa a expressão do *pointcut* para a transformação e a Figura 129 representa o *advice* da transformação. O código WAT resultante da transformação sem o modo inteligente está ilustrado na Figura 130, e o resultado com o modo inteligente está ilustrado na Figura 131.

```
(i32.add (call $f0) (call $f1))
```

Figura 127 - Código WAT original para o modo "inteligente"

```
() => call(* *(..))
```

Figura 128 - *Pointcut expression* para transformação no modo "inteligente"

```
(call $before (i32.const %call.Caller.Order%))
(target %this%)
(call $after (i32.const %call.Caller.Order%))
```

Figura 129 - Código do *advice* para a transformação no modo "inteligente"

```
(i32.add
  (call $before 0) (call $f0) (call $after 0)
  (call $before 1) (call $f1) (call $after 1)
) ;; Instrução incorreta
```

Figura 130 - Código WAT resultante sem o modo "inteligente" ativo

```
(call $before 0) (local.set $tmp0 (call $f0)) (call $after 0)
(call $before 1) (local.set $tmp1 (call $f1)) (call $after 1)
(i32.add
  (local.get $tmp0)
  (local.get $tmp1)
)  ;; Instrução correta
```

Figura 131 - Código WAT resultante com o modo "inteligente" ativo