Towards Language Parametric Web-Based Development Environments

Olaf Maas

Towards Language Parametric Web-Based Development Environments

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Olaf Maas
born in Eindhoven, The Netherlands

Programming Languages Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Towards Language Parametric Web-Based Development Environments

Author:	Olaf Maas
Student id:	4086864
Email:	thesis@maas.gr

Abstract

Language Workbenches are instruments developers use to create new domain-specific languages. They provide tools to rapidly develop, test and deploy new languages. Currently, workbenches support deployment in desktop-based integrated development environments. Setting up these environments can be a hurdle for the often non-technical users of these languages. Web-Based IDEs could be a solution in this case, but workbenches are currently not able to deploy languages in these environments.

This work presents the first step towards language workbenches in Web IDEs by creating a language parametric runtime for the browser which serves as a back-end for Spoofax. Combined with an editor, this runtime is the basis for the generation of entirely client-side language playgrounds based on Spoofax specifications. For parsing, this runtime has similar performance characteristics as the existing Spoofax implementation. Code execution in this runtime can be used in environments where performance is not critical.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. P. D. Mosses, Swansea University, UK
Committee Member:	Dr. J.S. Rellermeyer, Faculty EEMCS, TU Delft

Preface

A while ago I inquired about possible subjects for my Master Thesis at the Programming Languages group. Eelco returned the question ”What is your interest?”. Having just read the week before about a new format called WebAssembly, I replied that I would be interested in doing some research which would include that new format.

Our first idea was to get DynSem to work in WebAssembly. Little did I know at the time that this would lead to me building a language parametric runtime. I would like to thank Eelco for the opportunity and supervision of this project and the members of my committee for their time. Furthermore, I would like to thank all members of the PL group which I've met during the course of my thesis. Lastly, I would like to thank Willem Vaandrager, Frits Kastelein & Ri(c)k Nijessen for the coffee/tea breaks during my thesis in which I was able to clear my mind and of course my parents for supporting me throughout my study.

Olaf Maas
Delft, the Netherlands
July 6, 2018

Contents

List of Figures

Chapter 1

Introduction

Developing software is a complicated, expensive and error-prone process. With the size and complexity of applications rising, new layers of abstraction are necessary to abstract away this complexity. One way to create these new abstractions is the usage of domain-specific languages (DSLs). These are high-level, expressive programming languages which target a single problem domain. [10] Using this expressiveness developers and domain experts can express their idea without the need to care about low-level implementation details.

At its core, a new DSL consists of three components: a parser, transformer, and an execution environment. Though these three components create a functioning DSL, other services might be necessary. For example, modern developers expect these tools to work with their favorite integrated development environment and expect to get intelligent feedback.

Creating all these components from scratch requires a lot of effort. Due to their limited scope, DSLs are often created with little resources. Therefore, over the years many tools have been developed to simplify the process of developing these components. For example, parser generators help developers to generate their parser based on a high-level description. Term rewriting languages allow language developers to declare transformations on their abstract syntax tree. Executable semantic formalisms add a way for a developer to generate an execution environment based on a high-level specification of the semantics of their language. The creation of these tools decreased the resources necessary for the development of a new DSL.

Language Workbenches are frameworks which integrate a number of these tools in a single environment. With this integration they generate an environment where developers can quickly prototype and deploy their language, enabling them to do agile programming language development. Furthermore, they allow a developer only to write the unique parts of their language and let the workbench take care of the repetitive work such as IDE integration and parsing.

End-users of languages developed using these workbenches are often people with domain knowledge, not experienced developers. Setting up a development environment in a correct manner can be a challenge for these people, increasing the hurdle to start using a DSL. In this situation a web-based development environment could be a solution for these types of users: instead of having to set up their development environment they only have to open a web page to use a DSL.

Where language workbenches are a good fit when deploying languages in a desktop based environment, they often lack the support of deployment in web-based environments. This work improves support for web deployment of programming languages developed in the Spoofax language workbench [26]. We develop a new Spoofax back-end in the Rust programming language which can be compiled to the browser using WebAssembly. This new prototypical back-end can parse, desugar and execute Spoofax based languages entirely client-side.

1.1 Previous Work

Previous work ported parts of the Spoofax pipeline to the browser. The idea behind this research was to compile the existing Spoofax libraries (written in Java) using the Google Web Toolkit to JavaScript. GWT is Java to JavaScript cross-compilation framework which allows the development of web-based applications in Java. They also developed a Stratego to Javascript code generator which generates Javascript code based on a Stratego definition. These programs would behave like their Stratego counterparts when executed. [44]

Although the resulting artifacts were able to parse and transform programs, the ability to execute a piece of code was absent. Both the transformer and parser performed slower than their Java counterparts, and the compiled JavaScript libraries had a large binary footprint. Due to this footprint, the loading times of the editor were long. To increase the speed of the generated editor parts of the calculations were offloaded to a web server. [51]

This thesis improves upon this work in three ways, first of all, the parsing and transformation environment it delivers have performance characteristics which are more similar to the Java-based Spoofax implementation. Furthermore, we add the ability to execute programs entirely client-side. Finally, the delivered environment has a smaller footprint and lower initialization time.

1.2 Client-Side Code Execution

Client-side code execution is the concept of executing code provided by a web-page in the web-browser (client) rather than the server. In the early days of the web, this was often done using plug-ins such as Flash and Silverlight. With the rise of mobile browsers on tablets and smartphones these plug-ins became deprecated. Without these plug-ins, most browsers only support JavaScript as a language for the execution of code. For most web applications JavaScript is a good fit, but as a compilation target, it lacks in two ways. Firstly distributing large amounts of JavaScript creates a parsing overhead. Moreover, the dynamic and garbage collected nature of

JavaScript adds an unnecessary runtime overhead for lower level languages such as C and C++.

This unsuitability led to the creation of WebAssembly. WebAssembly is a portable and low-level binary format which is supported by all major browsers. Its goal is to be "a portable, size- and load-time-efficient binary format to serve as a compilation target which can be compiled to execute at native speed". [52][18] Currently WebAssembly is still in active development, therefore a number of features are still missing. Examples of these kinds of features are the lack of a standard library and access to the Document Object Model(DOM) without the use of the JavaScript API.

1.3 Web-based Development Environments

Recently more and larger applications which were once desktop-based applications found its way to the web. Software once only available on the desktop such as photo-editors, office suites, and media players are now available through the browser using web pages backed by the cloud. This move induced the start of a new generation of operating systems such as Chrome OS and Windows 10S, which are just a thin wrapper around a web browser. The surge of these operating systems shows that cloud-based solutions could become the norm for more everyday applications.

An Integrated Development Environment (IDE) is a piece of software developers use to develop software. IDEs often bundle a number of tools developers use during the development process. Using an IDE, a developer can edit, execute, debug and commit a piece of code without the need of switching applications. Just like other classic desktop applications, some development environments moved to the web, which introduced the concept of Web IDEs.

Today multiple Web IDEs exist ranging from commercial software (Cloud 9) to open source alternatives (Eclipse Che). At a glance, these web-based IDEs look similar to their desktop counterparts. However, existing plug-ins cannot be used directly in a Web IDE as it often requires parts of the compilation pipeline to work in the browser. Most workbenches, however, have a back-end which cannot be run in the browser.

But even if a workbench has a back-end which can run in the browser there are still a number of issues. We identify four main problems which still need to be solved:

- **Parsing & Highlighting** For performance reasons, Web IDEs do parsing and syntax highlighting in the browser. This client-side parsing introduces a problem for language workbenches as their generic parsing algorithms or code generators are not able to run directly in the browser without performance overhead.

- **Code Execution** Code written in an IDE needs to be executed at some point. Execution environments generated by language workbenches are often not suitable to run remotely out of the box, but they cannot run on the client either.

- **Performance** In the browser scripts run in a sandboxed environment and threading is limited. Languages ran using a language workbench often already have some performance overhead compared to a manual implementation. This overhead gets magnified when running in a web-based IDE which could result in a sluggish feel from an end-user perspective. As a developer is able to add caching and optimizations, for language-specific solutions running on the server might be an option. However, for generic algorithms used in language workbenches adding these optimizations might require changes in existing libraries.

- **Footprint** Artifacts generated by language workbenches often have a large binary footprint. Although this is not a big problem for local IDEs, in web-based IDEs loading 30 megabytes of artifacts is not acceptable.

To keep language workbenches, and by proxy languages developed using them, future proof it is important that eventually, they can interact with web-based IDEs. However, getting all components of a language workbench to work with a Web IDE at once is a large project. An intermediate step would be to get a subset of the components work in a smaller environment where performance is not critical.

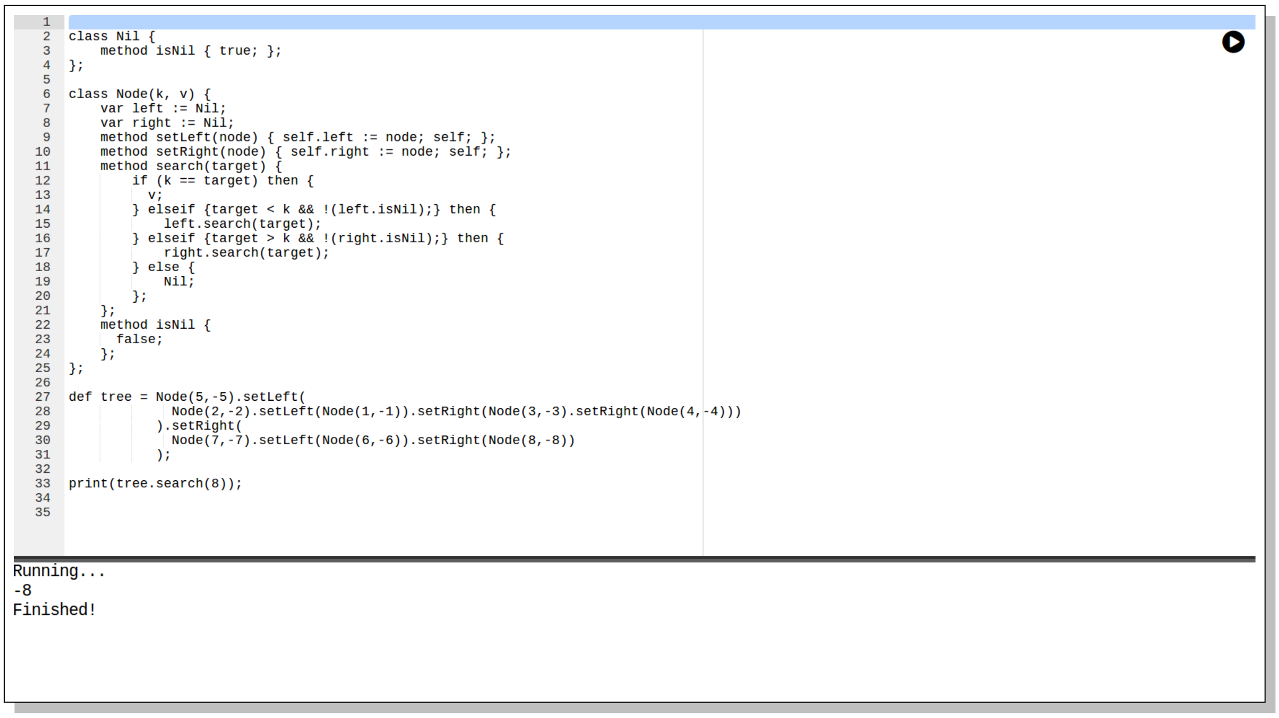
1.4 Generating Language Playgrounds

Language Playgrounds are simple code editors used to execute code snippets. They allow users to fiddle with the language in an online environment without having to install the language locally and are used in both the documentation as well as the promotion of a language. Although playgrounds are a simplified version of a web-based IDE, they still require most of the services a web-based IDE need such as parsing and execution.

In the process of enabling language workbenches to work with web-based development environments, generating a language playground could be a good intermediate step. Not all editor services are necessary and performance is not critical. For language developers, these playgrounds still could be a helpful tool in the promotion and documentation of their language.

Three components are necessary to generate language playgrounds based on the workbench definition of a language: a workbench runtime, code editor and terminal. JavaScript-based code editors and terminals already exist, so the only missing piece in playground generation is a web-based language workbench runtime. This runtime should be able to parse, transform and execute a program and print some output to the terminal. Just like in Web IDEs parsing should take place on the client-side to improve the editor's responsiveness.

Code execution could be done on both the client or the server. Existing language playgrounds execute code both on the client and the server-side. Despite server-side



```

class Nil {
    method isNil { true; };
};

class Node(k, v) {
    var left := Nil;
    var right := Nil;
    method setLeft(node) { self.left := node; self; };
    method setRight(node) { self.right := node; self; };
    method search(target) {
        if (k == target) then {
          v;
        } elseif {target < k && !(left.isNil);} then {
            left.search(target);
        } elseif {target > k && !(right.isNil);} then {
            right.search(target);
        } else {
            Nil;
        };
    };
    method isNil {
      false;
    };
};

def tree = Node(5,-5).setLeft(
             Node(2,-2).setLeft(Node(1,-1)).setRight(Node(3,-3).setRight(Node(4,-4)))
           ).setRight(
             Node(7,-7).setLeft(Node(6,-6)).setRight(Node(8,-8))
           );

print(tree.search(8));
```

```
Running...
-8
Finished!
```

Figure 1.1: Generated Language Playground for the Grace language

execution being faster, it requires a language developer to set up arbitrary code execution. Furthermore, it would require more resources. Although client-side code execution is slower, it is easier to deploy as it only requires some static web page. Therefore, a cheap host suffices which reduces the resources necessary for deploying such playground.

1.5 Objectives

Lack of support for web deployment of language parametric runtimes is blocking the path to language workbenches on the web. In an ideal world, deploying a language to the browser would only require a few mouse clicks. To achieve this ideal world, it is important that a language could be ported automatically to the browser.

This work its objective is to develop a language parametric runtime which can run in browser-based development environments. Such runtime should, based on the artifact generated by a language workbench, behave like the language defined using that workbench. Creating this runtime enables language developers to automatically port languages to the browser. Moreover, it improves the integratability of languages developed using workbenches in browser-based applications in general.

However, an automatic port is pointless when the cost of such port is too high for the developer. Cost in this context has multiple aspects. First of all, an automatic port should have a performance which is comparable to the performance in an offline environment. If this performance is not similar, the resulting development environment could feel sluggish. Initializing slowly or having to transfer high amounts of data could also have a negative effect on the resulting editor. Therefore the footprint of the runtime in both binary size as well as initialization time should be low.

To automatically port a language its parsing, transformation, and execution environment should be able to run in the browser. Spoofax has support for declaring each of these components. By porting or compiling the existing Spoofax back-ends to the browser, we would enable automatic porting.

Currently Spoofax its back-end is written in Java. Previous work showed that compiling this back-end to Javascript resulted in a parser which was slow and took long to initialize. Therefore it was not a perfect compilation target. The introduction of WebAssembly adds a new potential client-side compilation target. If we could port Spoofax its back-end algorithms to WebAssembly generating a language runtime which has performance characteristics more similar to the existing back-end might be possible.

To evaluate the runtime, we instantiate the runtime with three existing Spoofax languages and perform two benchmarks. First of all, the new runtime is compared to the existing runtime in terms of performance. Secondly, the performance overhead of running in WebAssebmly is measured by comparing the performance of the runtime ran natively against the runtime ran in the browser. Lastly, we measure the footprint of the new runtime in both initialization time as well as binary size.

1.6 Outline

In the remainder of this work, we dive into the development of *spfx_rs* a Rust port of Spoofax which can run in the browser. Chapter 2 describes a high-level architecture of this runtime. In Chapter 3 we describe the setup of the evaluation of *spfx_rs*. SGLR parsing, its implementation, and evaluation in our pipeline is explained in Chapter 4. Chapter 5 describes the DynSem metalanguage and its implementation in Rust. An overview of related work is given in Chapter 6. Chapter 7 proposes the next steps and possible future work. Finally, Chapter 8 reflects on the developed runtime.

Chapter 2

Architecture

spfx_rs is the language parametric language runtime which we developed. Such runtime can be seen as a program which, based on the specification of a language in Spoofax, behaves like a runtime for this particular language. For example, based on the specification of the Grace programming language this runtime acts as an interpreter for Grace. Before actually interpreting the provided program the runtime performs a number of steps, this chapter gives a high-level overview of these steps and presents an architecture of the developed runtime.

2.1 Rust

Rust is a young systems programming language focused on performance and memory safety. [21] Where most programming languages featuring memory safety achieve this by using a garbage collector, Rust accomplishes this by having a strict ownership model and borrow checker. Furthermore, it disallows shared state in combination with mutability between threads [37]. An extra advantage of Rust that it compiles to LLVM [31] which provides many compilation targets such as x86_64 on multiple platforms, ARM, and WebAssembly.

In the Rust ecosystem packages are called crates. They can depend on other crates and C(++) libraries. Cargo, Rust its build tool, manages these crate. During the build it resolves all dependencies by finding and downloading the correct versions of necessary dependencies. Besides being a dependency manager, Cargo also functions as a build tool. It can build, both libraries and executable binaries, and test projects. For these steps, it relies on the Rust compiler.

Rust is selected as the implementation language for the new back-end for three reasons. First of all, together with C and C++, it was one of the languages with usable WebAssembly support. C is eliminated due to its lack of memory safety, which left Rust and C++ as options. Rust was chosen for the good compiler support for preventing data races and having memory safety. Furthermore, Cargo as a package manager simplifies setting up a project, managing its dependencies and building it for multiple platforms. Lastly, the Rust community saw WebAssembly as a great

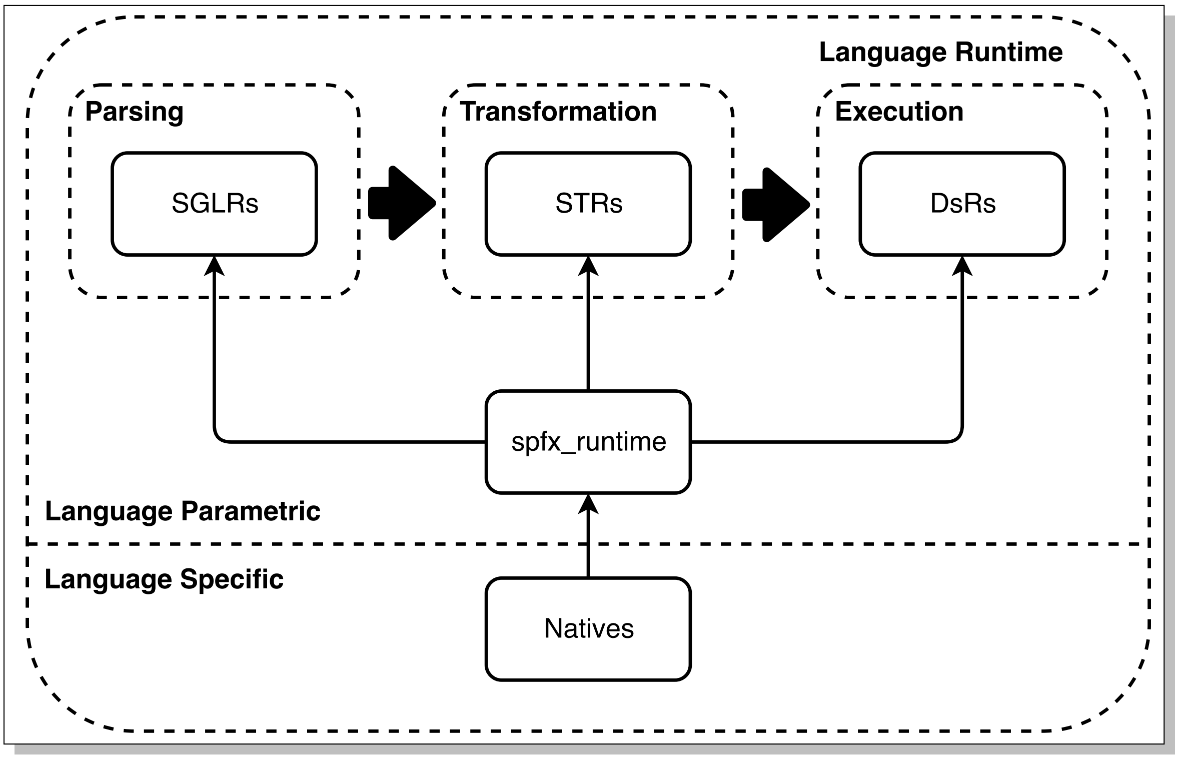


Figure 2.1: Architecture of *spfx_rs*

opportunity and showed a great commitment to keep improving this support.

2.2 Runtime Overview

Figure 2.1 shows an overview of the architecture of *spfx_rs*. At a high level, the runtime consists of two major parts: language parametric and specific crates. All crates which belong to the first part function at a meta-level, these crates can only be used when they are instantiated with a language specification. In the language specific part of the runtime the *spfx_runtime* crate is instantiated, this crate provides the APIs to communicate with all other components. Furthermore, it functions as the glue between all other steps of the pipeline. One of the components, DynSem, has the ability to call functions in its host language. As Rust is a statically typed language without the capabilities of runtime reflection, these operators need to be available at compile time. Therefore the language-specific crate contains the declaration of these operators.

2.2.1 Parsing

Before the interpreter starts execution, it needs to convert the plain text program to a structure on which it is better able to perform operations and analysis. Converting plain text to a data structure is called parsing and results in an Abstract Syntax Tree. Spoofax uses the Scannerless Generalized LR (SGLR) parsing algorithm for parsing. SGLR is a language agnostic parsing algorithm, which instantiated with a parse table of a language, behaves like a parser for this language [48]. Chapter 4 gives more information about SGLR parsing and its implementation in both Java and Rust.

2.2.2 Transformation

After parsing an AST is just a raw representation of a parsed program. To simplify and speed up interpretation languages often perform transformations to add more information to the AST. For example, certain constant expressions could be evaluated or more information about typing can be added to the AST.

Language developers declare rewritings in Spoofax using Stratego [50]. Stratego is the term rewriting language of Spoofax which traverses the AST. During these traversals, it matches certain AST nodes and rewrites them to new terms by (recursively) applying strategies. CTree is a lowered version of Stratego, which can be viewed as a bytecode-like format. Jeff Smits developed an interpreter for this format in Rust. *spfx_rs* uses this version in the Spoofax Rust pipeline. To support more languages, this version is extended with the ability to call user-defined strategies. These are a special type of strategy which the host language of Stratego provides.

2.2.3 Execution

After transformation, the AST is ready for interpretation. DynSem is the language used in Spoofax for the definition of the Dynamic Semantics of a programming language. Based on these specifications an interpreter is generated which recursively applies reductions to the AST to evaluate it [47]. Chapter 5 provides information about DynSem and the implementation of DynSem in Rust.

2.2.4 Other Services

Although the steps described in the previous sections are sufficient to execute some non-trivial languages, most languages require more steps before interpretation or compilation can be done. To this end language runtimes often contain more services such as name binding, type checking and debugging. Having these extra abilities improves the usability of a language runtime. In the future, the runtime could implement these features by either improving and extending existing crates, or the introduction of additional crates which provide these new capabilities.

2.3 Combining All Crates

To improve usability, the *spfx_runtime* crate contains all crates necessary for the execution of Spoofax based languages. Furthermore, this crate provides a factory

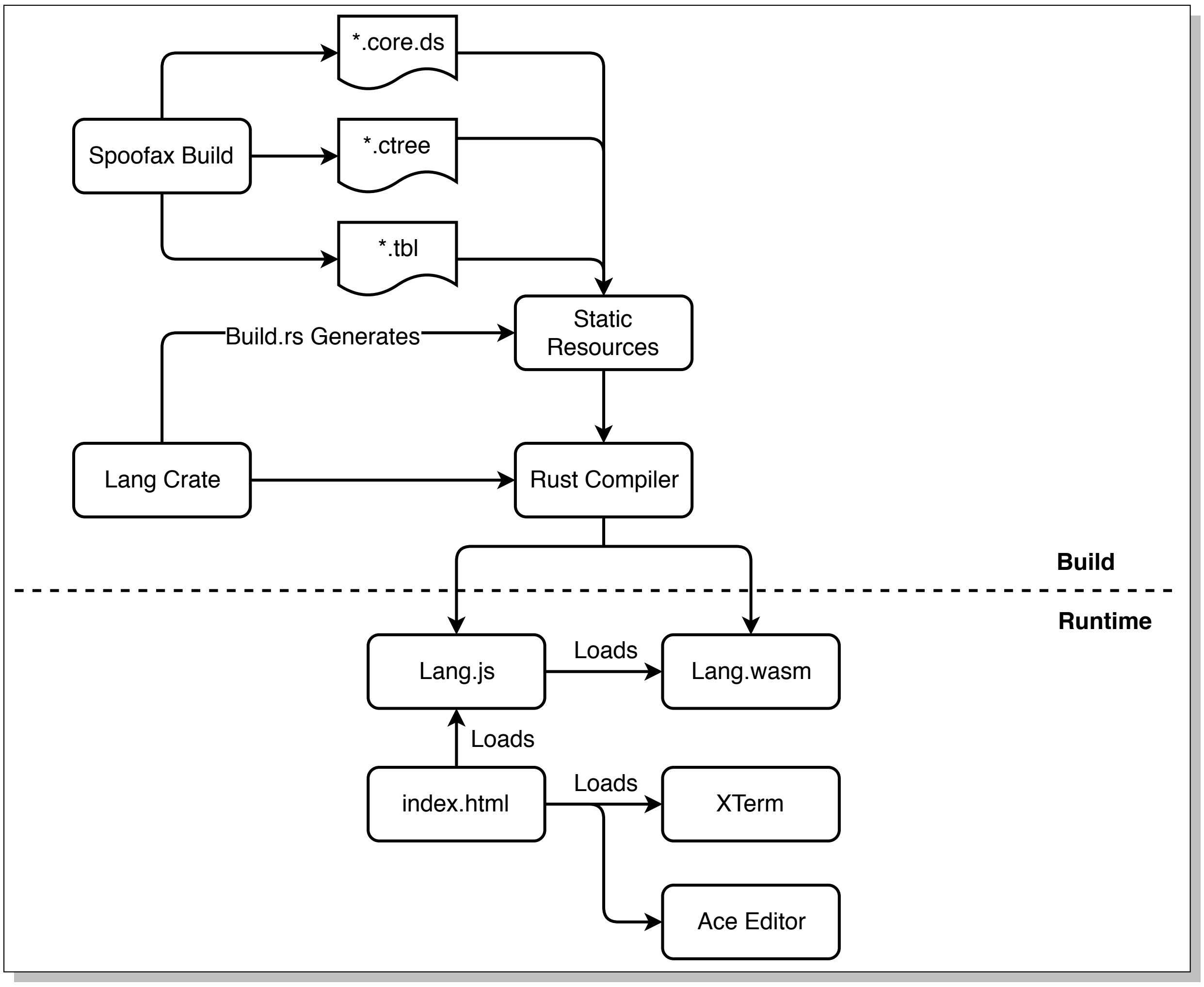


Figure 2.2: Build Process

which instantiates a new Spoofax runtime based on three arguments:

- **Parser** *spfx_runtime* contains two predefined parsers: 1) a literal ATerm parser, 2) the SGLR parser, which can be instantiated with a parse table. When these parsers do not suffice, or an existing handwritten parser exists a developer can provide a custom parser by implementing the Parser trait.

- **Execution Environment (optional)** Should be instantiated with a DynSem core file. For each native in this specification, a Rust function pointer should be passed to the instantiation function.

- **Rewriting Rules (optional)** Expects a file in the CTree format. When the provided CTree program uses native strategies, the language developer should

```
pub fn eval(
  program : String,
  core_location: String,
  tbl_path: String,
  list: NativeList,
  tbl_file: String
) {
    let parser = DefaultParser::new();
    parser.with_parse_table(tbl_path);

    let mut runtime =
    LanguageRuntimeFactory::<DefaultParser>::new("Lang_name"
       , &parser)
      .with_core(core_location, native_list)
      .create();

    runtime.run(program)

}
```

Figure 2.3: Runtime initialization in Rust

provide a mapping from strategies to functions during the initialization of the Stratego interpreter.

An example of the code necessary for instantiating a language runtime can be found in figure 2.3.

2.4 Compiling to WebAssembly

With the crates described in the previous section, one is able to run the code natively. Compiling this code to WebAssembly and deploying it in the browser require some extra steps. This section gives an overview of how the Rust crate is compiled to WebAssembly and how JavaScript communicates with Rust. Lastly, it shows how this code is deployed.

2.4.1 Compilation

Rust compiles to WebAssembly in two ways. *wasm32-unknown-emscripten* is the classic way and requires a version of Emscripten [55], an LLVM to JavaScript/WebAssembly compiler, to be installed on the host system. Rust passes the LLVM-IR code which it emits to the Emscripten compiler, which does the linking and compiles it to WebAssembly. During this process, Emscripten adds certain standard

```
#[derive(Serialize,
    Deserialize)]
pub struct Dog {
  name: String,
  age: usize
}

let dog = Dog
  {name: ""Foo, age: 10};

js!(
  var dog = @{dog};
  console.log(dog.name)
)
```

```
pub fn new_dog(
  name: String,
  age: usize
) -> usize {
  ...
  unsafe{
    transmute(Box::new(dog));
  }
}

pub fn bark(
  dog_pointer: usize
) {
  let dog : &Dog = unsafe {
    *dog_pointer
  }
  ...
}
```

Figure 2.4: Rust JavaScript Interop with (left) and without (right) serialization

library functions such as a virtual file system, standard output, time and memory allocation.

Recently Rust added support for compilation using the official LLVM WebAssembly backend. This target is called *wasm32-unknown-unknown*. By introducing this target Rust removes its dependency on the Emscripten compiler for WebAssembly support. In the long run, this should stabilize WebAssembly support. Furthermore, this compiler produces error messages at a higher compilation level which results in clearer and more precise error messages. One of the drawbacks of this method is that it only offers rudimentary standard library support.

2.4.2 Rust/JavaScript Interoperability

For the integration of a Rust library in a web application, interoperability between Rust and JavaScript is necessary. In the compilation of *spfx_rs* two tools have an important role: Cargo web and std_web. Cargo Web [1] is an extension for cargo which adds functionality to quickly build, test and deploy WebAssembly builds done using the Rust compiler. It contains a simple server which will serve all files necessary to run the result of a WebAssembly build.

Std_web is a crate which improves JavaScript and Rust interoperability by exposing a number of Web APIs through a Rust library. It allows Rust to access the Document Object Model and includes the ability to directly execute snippets of JavaScript from Rust code. Combined with Cargo Web, std_web adds the ability to use Rust libraries like pure JavaScript libraries.

During execution, passing non-primitive types between JavaScipt and Rust could be necessary. *std_web* provides a method to share objects between JavaScript and WebAssembly. Figure 2.4 show an example of this. In this example, Rust passes the Dog struct to JavaScript and executes this snippet resulting in the logging of the dog its name. In the background, *std_web* will serialize the object to a JavaScript Serialized Object Notation (JSON) string, a format JavaScript is able to understand. For small structures this step does not have a big performance impact, for bigger structures, like an SGLR parse table, this could become a performance bottleneck.

When it is necessary to pass larger structures between Rust and JavaScript, we transfer the structure's ownership to JavaScript and return a pointer to the structure on the WebAssembly heap to JavaScript. Methods which accept this pointer are used to do operations on this object. Downside of this solution is that it requires transmuting memory which is an unsafe operation in Rust. Although this solution removes some safety guarantees, the solution is faster than serializing and deserializing for bigger data structures.

2.4.3 Deployment

For web standards the artifacts Spoofax generate are large. For example, the table SGLR uses for Grace is ± 2 megabytes big. All CTree libraries which are necessary to lower Grace are ± 5 megabyte in size. Lastly, the .core file of the DynSem specification adds a megabyte. All these artifacts are required to instantiate the runtime, and without them, no program can be executed.

Before initializing of the runtime, the browser should fetch all artifacts. One option would be to load all files separately when the page loads. By using this option, the browser can cache the files at an individual level. This caching would result in a reload of only the changed artifacts when the definition of the language changes. A downside of this solution is that WebAssembly cannot load files without JavaScript. Loading the artifacts through JavaScript would require passing large Strings from JavaScript to Rust, which could become expensive.

To decrease the number of large structures passed between WebAssembly and JavaScript we use some code generation. Before actually building the project Cargo bundles all artifacts into the WebAssembly binary using a build script. This script generates a Rust file which contains the contents of all the Spoofax artifacts in the form of raw String literals. A downside of this approach is that when one artifact changes all other artifacts are reloaded as well.

By integrating the resulting binary in an existing HTML page, the deployment process can generate a language playground style environment. This environment consists of an Ace based editor, a terminal emulator and a play button. For terminal emulation we use XTerm.js. When a user presses the play button a function is activated which passes the program from the editor to the Spoofax Runtime. All output created by DynSem and the runtime is passed to the XTerm.js.

Chapter 3

Evaluation Setup

As an evaluation, we compare the new back-end to the existing one. This chapter describes what methods and measurements we used to make this comparison. The first section gives an overview of the languages which were selected for in the benchmark. Section 2 describes the platforms used in the comparison. Lastly, an overview of the measurements and measuring instruments is given.

3.1 Benchmark Languages

Benchmarking a meta-runtime requires a language implementation to instantiate the runtime's components with. As instantiators, three benchmark languages are selected. Two selection criteria are used: 1) An existing Spoofax specification for the language should be available, 2) A working DynSem specification is included in this specification. By using these criteria the development time necessary to set up the benchmark suite is reduced. Furthermore, these are actual Spoofax implementations developed by external developers, which should result in more realistic benchmark results.

```
function fib(n) {
  if (n < 2) {
    return n;
  } else {
    return
      fib(n-1) + fib(n-2);
  }
}
```

```
let
  function fib(n : int) :
     int =
   if n > 0 then
     n
   else
     fib(n-1) + fib(n-2)
in
  fib(12)
end
```

Figure 3.1: Fibonacci function in SL (left) and Tiger (right)

```
class Node(next, content) {
  var next := next;
  var content := content;
};

def list = Node(
  Node(Nil, 19), 3
);
```

```
let
  type Node = {next: node,
     content: int};
in
  Node{left = Node{
    left = nil, content=19
  }, 3}
end
```

Figure 3.2: Singly Linked List data structure in Grace(left) and Tiger(right)

Based on these requirements the following three languages are chosen.

1. **SL** is a basic imperative language consisting of function, objects & control flow statements such as loops. To allow easier execution of some benchmarks in the suite SL is extended with array support. For this purpose, the parts of the existing Object system are reused. Note that SL is the only DynSem based specification which uses the Native Data Type system.

2. **Tiger** is an imperative language which has support for records, let bindings functions and arrays. Andrew Appel uses Tiger as an example language in its book "Modern Compiler Implementation in Java". [4] As part of the Compiler Construction course at the Delft University of Technology, which uses this book, a specification in Spoofax was created.

3. **Grace** is an object-oriented programming language designed with education in mind. Its main goal is that it should be easily teachable to University students while integrating proven ideas other programming languages uses. [7] In Grace objects can be created without the need of a class, fields of an object are immutable. Classes describe the structure of an object and can implement a trait. Both mutable and immutable bindings are supported by Grace. Furthermore, Grace has support for imports and dialects of the language. As part of a master thesis at the Delft University of Technology a Spoofax implementation of Grace was created [19] The DynSem specification of Grace interprets a lowered version of the parsed AST. In Spoofax this lowering is done by Stratego.

As an example, and to give an idea how these languages look, a number of code snippets are provided for these languages. An example of a recursive function which calculates the first n Fibonacci numbers implemented in both SL & Tiger can be found in figure 3.1. Both Tiger and Grace support objects based on templates. Figure 3.2 shows how to implement a basic linked list in Grace and Tiger.

3.2 Benchmark Setup

Each platform is instantiated with a language, then six different benchmarking programs are run on this platform. The benchmark set consists of programs which test multiple aspects which are typically encountered in the context of a language playground: smaller problems where string operations, integer arithmetic and data structure creation are often used in combination with some iteration. As input for the benchmark, a benchmark set consisting of the following programs was used:

1. **Fib40** Calculate the 40th number of a Fibonacci sequence using an iterative approach.

2. **5Queens** is a program which solves the problem of placing 5 queens on a 5*5 chessboard, with the requirement that no pair of queens can attack each other. It uses a backtracking algorithm.

3. **BinTreeLookup** Create a binary tree of 10 elements. Lookup and print each element of the binary tree.

4. **Matrix10** Multiplying two 10x10 matrices using a naive algorithm which iterates over both matrices.

5. **Print800** is a program which prints ”Hello World” 800 times.

6. **Concat800** Concatenates 800 Strings.

Results are obtained through a benchmark tool which is written in Rust. This tool is responsible for spinning up benchmark platforms, sending a program which should be executed on this platform and obtaining the benchmark results. Note that all Rust targets are compiled in Release mode. The following platforms are compared using the benchmark tool:

1. **Java**: Spoofax libraries ran on the default JVM. Java its Just In Time compiler is enabled, this VM does not utilize features provided by the Truffle/Graal framework.

2. **GraalJava**: Spoofax libraries ran on the GraalVM. Graal is a highly specialized compiler which can convert simple tree-based interpreters written using the Truffle framework to high-performance JIT compilers. [54] During parsing and preprocessing the capabilities of Graal should not make a difference. For the interpretation step, however, Graal its capabilities should make an impact.

3. **Rust Native**: Native Rust builds the project using the default Rust stack size. Compilation is done using the nightly Rust compiler of 5th of February 2018. This target is included to compare native execution speed to WebAssembly performance.

4. **Rust Wasm**: Rust WebAssembly build done using the nightly Rust compiler of 5th of February 2018 with the *wasm32-unknown-unknown* target. Programs are run in a Headless Chrome instance which is controlled by the puppeteer framework.

5. **SLMan**: Tree-based interpreter for the SL language which is written in Rust. Note that this is a manual port. For parsing, the SGLR parser is used. The semantics of SL however are all written in Rust code. Size of the codebase is 600 loc.

Benchmarking in previous work was done using an older Spoofax version which was compiled with an unknown version of the Google Web Toolkit, therefore we are not able to reproduce these results. This means that the results of our benchmark cannot be directly compared to the results generated by the previous work. Figures provided in the previous work can only be used for comparison, but not as an absolute benchmark.

3.3 Measurements

3.3.1 Speed

Execution Speed is an important part of the user experience when using a web-based IDE. To get an idea how big the runtime overhead is for automatically porting a language, the Rust implementation of Spoofax is compared to the Java-based Spoofax back-end. This comparison is done on both execution speed as well as parsing speed. All benchmarks are executed on a machine with an Intel i5-6200U CPU running on a frequency of 2.30 Ghz with 8 gigabytes of memory. It runs Elementary OS Loki 4.1, a Linux distribution based on Ubuntu 16.04.

WebAssembly benchmarks are executed in a headless Firefox web browser. Its version is 58.0.2 for Ubuntu, and all plugins are disabled during benchmarking. WebDriverIO [53], a Javascript front-end for Selenium [40], commands this instance and navigates it to a benchmark HTML page which is served by a local web server. After all modules and initializations are done a flag is set, and the benchmarks are executed. Due to safety concerns, browsers do not expose an exact performance API, for the benchmark programs this does not matter as they are not in the nanosecond precision.

Native Rust benchmarks are run using a Rust program which initializes a language runtime and then executes all benchmark programs ten times. Compilation is done using the nightly Rust compiler of 5th of February 2018. Rust its default stack size is used. Elapsed time is measured using the precise time ns functionality of Rust which provides a fine enough granularity for the benchmarks.

Java benchmarks are done on both a vanilla JVM and the GraalVM. For benchmarking the Java Microbenchmark Harness (JMH), a benchmarking framework is used. This framework adds the ability to set up a benchmark without measuring the setup time. Before measurement JMH inserts a number of warmup rounds, this removes the noise introduced by the JVM optimizing certain hotspots.

3.3.2 Binary Size and Initialization Time

In general, it is a desirable feature for web applications to have a quick initialization time. To keep this time low it is necessary for the runtime to have a low binary size, so it can be quickly initialized after loading the page. Therefore both binary size and total initialization times are measured for web-based targets. For binary size, all files necessary to run a language are taken into account. We measure initialization time by measuring the amount of time necessary from the moment a document is ready until the page can execute programs. The size of a language runtime should not exceed the size of a large popular website like Facebook or Google. Initialization time of a language should not exceed the amount of time it takes to load a video, 5 seconds at most.

3.3.3 Other Factors

When the cost of a manual port is comparable to the cost of an automatic port, a developer might decide to do the manual port. Therefore we compare the cost of a manual port to the cost of an automatic port. Porting time and performance are obvious factors in this comparison. Adding features to a language when multiple back-ends exist can become expensive as both back-ends need to be altered. Therefore we also include maintenance cost in this comparison.

Chapter 4

SGLR in Rust

Parsing is the first step in the process of executing a language. It transforms plain text to an abstract syntax tree (AST). Spoofax-based languages use JSGLR, a Java implementation of the Scannerless Generalized LR (SGLR) parsing algorithm to parse files. This chapter presents *SGLRs* an implementation of the SGLR algorithm in Rust. First, it provides an introduction to SDF and SGLR, then it gives an overview SGLR in Rust.

4.1 SDF

Spoofax uses the SDF3 metalanguage for grammar specification. SDF3 allows language developers to specify the grammar for their programming language in a BNF-like syntax. Besides specifying the grammar of the language, a developer can declare disambiguations and context-free priorities using SDF3. Based on this specification SDF3 generates a parser with auto-completion and error recovery, pretty printer and the AST constructors.

During the language build, Spoofax transforms SDF3 to the classic Syntax Definition Formalism. [49] This step resolves the modules of the SDF3 specification and insert rules and priorities to enable more advanced parser features such as error recovery and auto-completion. Furthermore, it generates the constructors which will be used in the rest of the pipeline.

Based on this lowered SDF specification, Spoofax generates a parse table for the SGLR algorithm. SGLR is a parsing algorithm which based on a parse table of a language, behaves like the parser for this language. [48] Note that part of the disambiguations can be done by the parse table generator, however, some remaining disambiguations are done during and after parsing using preferences and context-free priorities.

4.2 Scannerless Generalized LR Parsing

LR is a parsing algorithm developed by Knuth et al. [28] which produces a right-most derivation of a language in linear time. LR relies on a parse table, which is a finite state automaton represented as a table. This table tells LR what actions to execute next based on the stack top and the state of the automaton. For certain languages, LR parsers end up in a position where they, based on the state, can perform two actions. These situations are called shift/reduce conflicts and they limit the set of languages LR can parse.

Algorithm 1 SGLR Parse function

```
function PARSE(table, program)
    global accepting stacks ← $\emptyset$
    global active-stacks ← { new stack with state init(table)}

    do
        global current-token ← get-next-char(file)
        PARSE-CHARACTER()
        SHIFTER()
    while current-token $\neq$ EOF $\wedge$ active-stacks $\neq \emptyset$

    if accepting-stack contains a link to the inital stack with tree t then
        return t
    else
        return parse-error
    end if
end function
```

Generalized LR parsing [46] solves these shift/reduce conflicts by shifting and reducing at the same time when it encounters a shift or reduce conflict. Executing multiple actions results in two diverging parse paths. In the subsequent steps, when one of the paths cannot progress GLR discards this path. When no paths remain, GLR returns a parse error. Executing multiple parse paths might result in two paths which both end up in an accepting state. In this case, an ambiguity in the grammar is found and GLR will return both paths as separate parse trees.

Spoofax uses the Scannerless Generalized LR (SGLR) parsing algorithm. SGLR is an improvement over the GLR algorithm in two ways. Firstly, it adds reject productions to the GLR algorithm which when included in a parse tree remove this tree from the possible parse trees. These productions enable lexical disambiguation of possible ambiguous grammars. Secondly, SGLR functions on the character stream instead of a token stream, removing the need for a scanner.

Algorithm 1, 2, 3, 4, 5 and 6 give an overview of the SGLR parsing algorithm defined by Visser in 1997 [48]. It represents the parse stack as a graph. Vertices of this graph are states and edges of this graph are parse trees. SGLR keeps a list with

Algorithm 2 Parser function

```
1:  function PARSE-CHARACTER
2:      global for-actor ← active-stacks
3:      global for-actor-delayed ← $\emptyset$
4:      global for-shifter ← $\emptyset$
5:
6:      while for-actor $\neq \emptyset \wedge$ for-actor-delayed $\neq \emptyset$ do
7:          if for-actor $= \emptyset$ then
8:              for-actor ← {pop(for-actor-delayed)}
9:          end if
10:         for each stack st $\in$ for-actor do
11:             if $\neg$ all links of stack st rejected then
12:                 ACTOR(st)
13:             end if
14:         end for
15:     end while
16: end function
```

the tops of active stacks which are alive.

One iteration of the SGLR algorithm consists of: fetching the next token, execute all actions possible based on the current stacks and this token, execute all shifts. Shifting in SGLR progresses the active stack tops to the next round of parsing. SGLR discards stacks in the active stack list which cannot shift as they are invalid parsing paths. During the execute action step SGLR looks up the next step for each active stack based on its state in combination with the current token. When it encounters an accept, it adds the stack to the list of accepting stacks. If SGLR encounters a shift, it adds this shift to the list of possible shifts which it executes after the reduction phase.

For a REDUCE action with production $\alpha \leftarrow A$ the algorithm finds all paths to the current state with length $|\alpha \leftarrow A|$. For each path p of these paths, the algorithm performs the actual reduction. SGLR performs a reduction for each different path as multiple paths could have led to the same state. A reduction results in a new state s based on a GOTO action and the stack top after reduction st. During reduction, the algorithm builds a new parse tree. This tree has as a root node a new node with the name of the production. Its children are the trees on the edges of the path the reduction is based on.

If a stack with state s already exists in the active stacks and there is a link between st and this stack, the tree on this link is changed to an ambiguity node. Else if a stack with state s exists and no link exists between this stack and st, a new link with the tree is created between st and the existing stack. Introducing this new link might add new paths to stacks which SGLR considered in earlier iterations of the parse-character loop. Therefore when it adds a new link, SGLR reconsiders the reduction of all stacks considered in previous iterations for this new link. If no stack with state *s* exists SGLR creates a new stack and adds a connection between the stack top and the new stack with the tree of the path on its edge.

Algorithm 3 Actor Function

```
1: function ACTOR($st$)
2:     for each action $a \in$ actions($s$, $current\text{-}token$) do
3:         switch $a$ do
4:             case shift($s$)
5:                 $for\text{-}shifter \leftarrow \{\langle st, s\rangle\} \cup for\text{-}shifter$
6:             case reduce($\alpha \rightarrow A$)
7:                 DO-REDUCTIONS(st, $\alpha \rightarrow A$)
8:             case accept
9:                 $accepting\text{-}stack \leftarrow st$
10:    end for
11: end function
```

Algorithm 4 Reductions Function

```
1: function DO-REDUCTIONS($st$, $\alpha \rightarrow A$)
2:     for each path from stack $st$ to $st_0$ from length $|\alpha|$ do
3:         $kids \leftarrow$ the trees of the links which form the path from $st$ to $st$
4:         REDUCER($st_0$, goto(state($st_0$), $\alpha \rightarrow A$), $\alpha \rightarrow A$, $kids$)
5:     end for
6: end function
```

Algorithm 5 Limited Reductions Function

```
1: function DO-LIMITED-REDUCTIONS($st$, $\alpha \rightarrow A$, l)
2:     for each path from stack $st$ to $st_0$ of length $|\alpha|$ going through link $l$ do
3:         $kids \leftarrow$ the trees of the links which form the path from $st$ to $st$
4:         REDUCER($st_0$, goto(state($st_0$), $\alpha \rightarrow A$), $\alpha \rightarrow A$, $kids$)
5:     end for
6: end function
```

4.3 SGLR in Rust

SGLRs is an SGLR implementation based on Rust. It takes a program and parse table as input and produces an ATerm which is used by the rest of the Spoofax pipeline. This section explains the differences between the Java and Rust version and gives an overview of this implementation.

Algorithm 6 Shifter Function

```
function SHIFTER($st$, $\alpha \to A$, l)
    $active\text{-}stacks \leftarrow \emptyset$
    $t \leftarrow$ current-token
    for each $\{\langle st, s\rangle\} \in for\text{-}shifter$ do
        if $\exists st_1 \in active\text{-}stacks$: state($st_1$) $= s$ then
            add a link from $st_1$ to $st_0$ with tree $t$
        else
            $st_1 \leftarrow$ new stack with state s
            add a link from $st_1$ to $st_0$ with tree $t$
            $active\text{-}stacks \leftarrow \{st_1\} \cup active\text{-}stacks$
        end if
    end for
end function
```

Algorithm 7 Reducer Function

```
function REDUCER($st_0$, $s$, $\alpha \to A$, kids)
    $t \leftarrow$ application of $\alpha \to A$ to $kids$
    if $\exists st_1 \in active\text{-}stacks$ : state($st_1$) $= s$ then
        if $\exists$ a direct link $nl$ from $st_1$ to $st_0$ then
            add $t$ to the possibilities of the ambiguity node at tree(nl)
            if $\alpha \to A$ is a reject production then mark link $nl$ as rejected
        else
            add a link nl from $st_1$ to $st_0$ with tree $t$
            if $\alpha \to A$ is a reject production then mark link $nl$ as rejected
            for each $st_2 \in active\text{-}stacks$ do
                DO-LIMITED-REDUCTIONS($st_2$, $\alpha \to A$, $nl$)
            end for
        end if
    else
        $st_1 \leftarrow$ new stack with state $s$
        add a link $nl$ from $st_1$ to $st_0$ with tree $t$
        $active\text{-}stacks \leftarrow \{st_1\} \cup active\text{-}stacks$
        if rejectable(state($st_1$)) then
            $for\text{-}actor\text{-}delayed \leftarrow$ push($st_1$, $for\text{-}actor\text{-}delayed$
        else
            $for\text{-}actor \leftarrow \{st_1\} \cup for\text{-}actor\text{-}delayed$
        end if
        if $\alpha \to A$ is a reject production then mark link $nl$ as rejected
    end if
end function
```

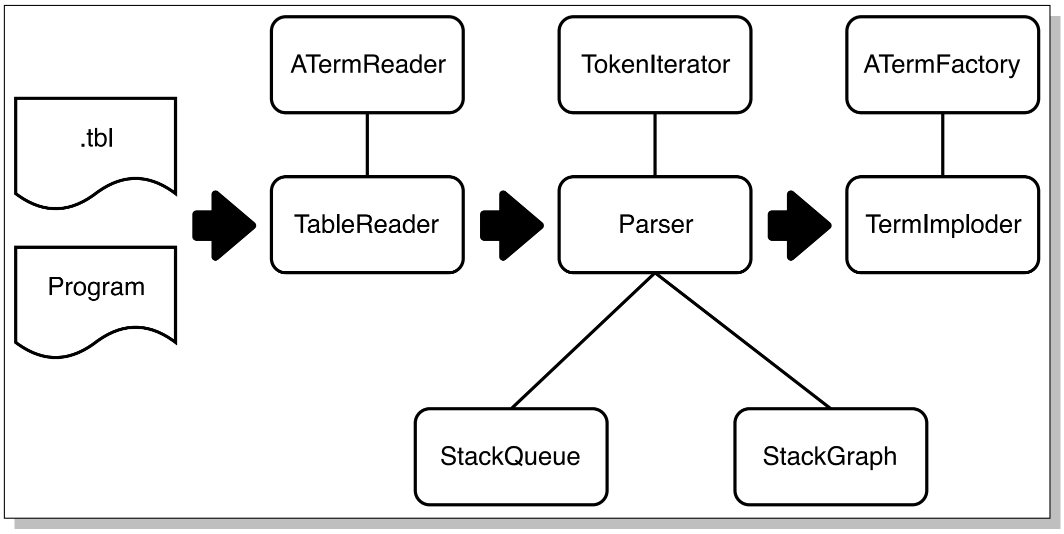


Figure 4.1: Architecture of the parsing crate

4.3.1 Architecture

Figure 4.1 shows the high-level architecture of the *SGLRs* crate. As input, this crate takes two parameters: a *.tbl* file representing a parse table generated by Spoofax and a program in the guest language. Before parsing the parse table reader reads the .tbl format and converts it to a Rust structure. Then the parser builds the parse forest. Lastly, the TermImploder implodes the parse forest into an ATerm which the next steps in the execution use.

During the parsing step, the parser interacts with three main components. The StackQueue keeps track of the stacks which are still alive. StackGraph is the component which keeps track of the whole parse graph and executes the path finding for the parser. Lastly, the TokenIterator is the component which keeps track of the position of the input stream and creates lookahead sets.

4.3.2 Parse Table

Since SGLR uses the parse table to determine which steps to perform next it is important that the algorithm is able to quickly look up gotos and actions. To put the parse table in a format which allows this quick lookup *SGLRs* preprocesses the parse table. SDF outputs a parse table (.tbl) file which *SGLRs* uses as input for the *SGLR* algorithm. A .tbl file consists of a stringified version of an ATerm representing a parse table. It contains the production rules and states. For each state, it also includes the goto and actions which should be performed. For disambiguation purposes, it also stores the priorities of the production rules.

SGLRs parses this table to a Rust structure which represents an ATerm. It transforms this format to a parse table structure and generates Gotos and Action maps so that based on a label or character identifier the algorithm can quickly look up which actions need to be performed. Note that the parse table contains some duplicate information, this step removes this duplicate information.

4.3.3 Algorithm

SGLR has multiple implementations and iterations of the algorithm in different languages. Spoofax uses *JSGLR*, the Java implementation of SGLR. Compared to the original SGLR algorithm this version adds support for more advanced features such as auto-completion, syntax highlighting an error recovery. To support these features, SDF inserts more production rules and terms into the parse table. *SGLRs* does not support these features, therefore it ignores these terms and productions.

TokenIterator is the service which provides the tokens to the SGLR algorithm. Although it names suggest that it is an actual iterator it is not an iterator in the classical sense as it gives the ability to do lookahead in the iterator without changing the result of the next element. SDF generates a parse table which assumes that characters are ASCII characters. Rust characters, however, are based on UTF-8 encoding. As the first 255 characters of ASCII correspond with the first 255 characters of UTF-8, this encoding difference does not pose a problem an no conversion is necessary.

SGLR keeps track of what it should do in the next step by maintaining a couple of global lists. In Rust, we pass around these lists in the algorithm and do not keep them globally. For this purpose the StackQueue struct exists, it is the owner of the *for-actor*, *for-actor-delayed* or *for-shifter* lists. Based on a reference to a list it can check whether this element is in one of these lists.

While parsing ambiguous grammars, SGLRs explores multiple parse forests when it encounters a shift-reduce conflict. Exploring multiple paths results in a parse stack which behaves like a graph. StackGraph is the Rust data structure which represents the parse stack as a graph. It manages the creation of new states and links between states and the pathfinding between this graph. Under the hood, it uses the *petgraph* crate which is a graph library for Rust.

At the end of the parsing step, SGLR returns a parse forest. This forest may still contain ambiguities and superfluous constructors. In the imploding step, these are removed by the imploder, a structure which implodes the parse forest created in the parse phase. It ignores terms in the resulting parse tree which are used for syntax highlighting. This step results in an Annotated Term representing the abstract syntax tree.

During imploding the imploder might encounter ambiguities. In this case, the imploder calls the disambiguator to disambiguate the forest. *SGLRs* supports only basic forms of disambiguation in the form of avoid, reject and prefer productions. These priorities are stored in the forest. During disambiguation, the disambiguator will first return all productions with the prefer attribute, when none are present it returns all normal productions, when these are not present as well it returns the avoid productions. Note that after disambiguation it is still possible that two terms remain. In this case, the disambiguator returns an ambiguity node.

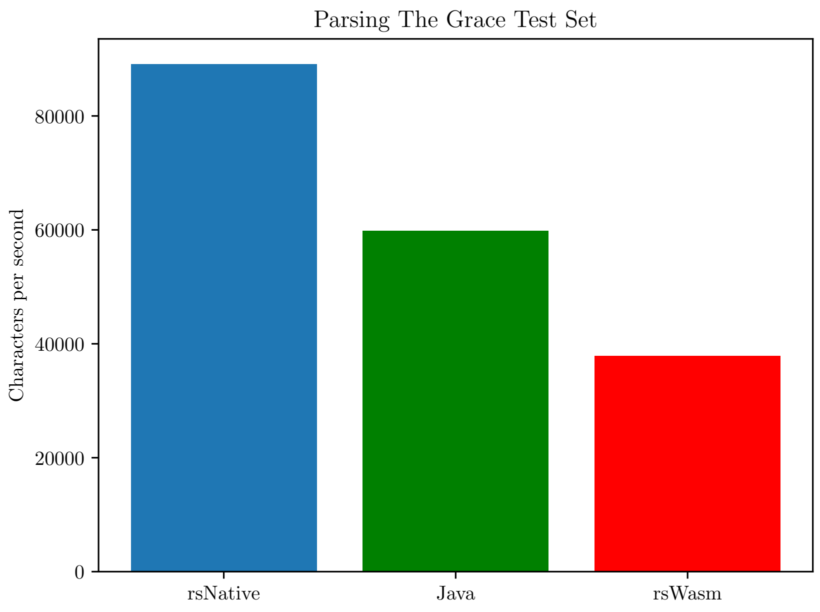


Figure 4.2: Parsing Speed for Grace lang

4.4 Evaluation

To evaluate the performance of *SGLRs* we compare it against the Java implementation of SGLR on the Grace test set. This set consists of 254 Grace programs ranging in size. Note that for the Java implementation error recovery was disabled. For Java benchmarking, JMH was used and benchmarks were executed on a normal JVM on the same machine described in the evaluation chapter.

Figure 4.2 shows the results for this benchmark. Run natively, the Rust implementation is faster than the Java implementation. WebAssembly shows an overhead. In practice, however, these parsing speeds are quick enough to be used for client-side parsing in a web-based IDE where only a few files are open at the time.

Compared to JSGLR, SGLRs is faster in parsing the Grace benchmark set. An explanation for this difference is that SGLR parsing is a relatively low-level, procedural operation. Except for the parse table and the input, most information about the algorithm is known at compile time which allows the Rust compiler and LLVM to do the necessary optimizations.

Chapter 5

DynSem in Rust

DsRs is an implementation of the DynSem meta-interpreter in Rust. Based on a DynSem core specification and an AST of the guest language this interpreter behaves like an interpreter for the guest language. This chapter provides an overview of this crate and DynSem. It compares the differences in implementation and explains the reasoning behind these differences. Lastly, this chapter presents the evaluation steps which were executed.

5.1 Overview

DynSem is Spoofax its meta-DSL for the specification of the operational semantics of a programming language. It enables a language designer to define the semantics of his language concisely and at a high level. Based on this specification DynSem generates a Java interpreter for the specified language.

During this generation step, the interpreter generator performs some steps. First of all, it resolves all modules of a DynSem specification and combines the different modules into one specification. Then the generator explicates the DynSem specification and saves it as a textual representation of the abstract syntax tree. This format is called the DynSem core format. For all defined constructors, (factory) classes are generated, these classes are loaded into a more generic meta-interpreter which executes the actual interpretation.

DsRs is an alternative back-end for DynSem which can be used to execute languages on the web and in a native environment. In contrary to the original, this new back-end is an interpreter. Initially, the plan was that *DsRs* would execute all transformation steps in Rust. This would add the ability do in-browser DynSem development. The existing back-end declares all these transformations as term rewrites in Stratego. Rust, however, is not very suitable for these type of transformations, and the decision was made to perform DynSem core interpretation. Note that by parsing DynSem and doing all lowering using *STRs* in-browser DynSem development in the browser is still possible.

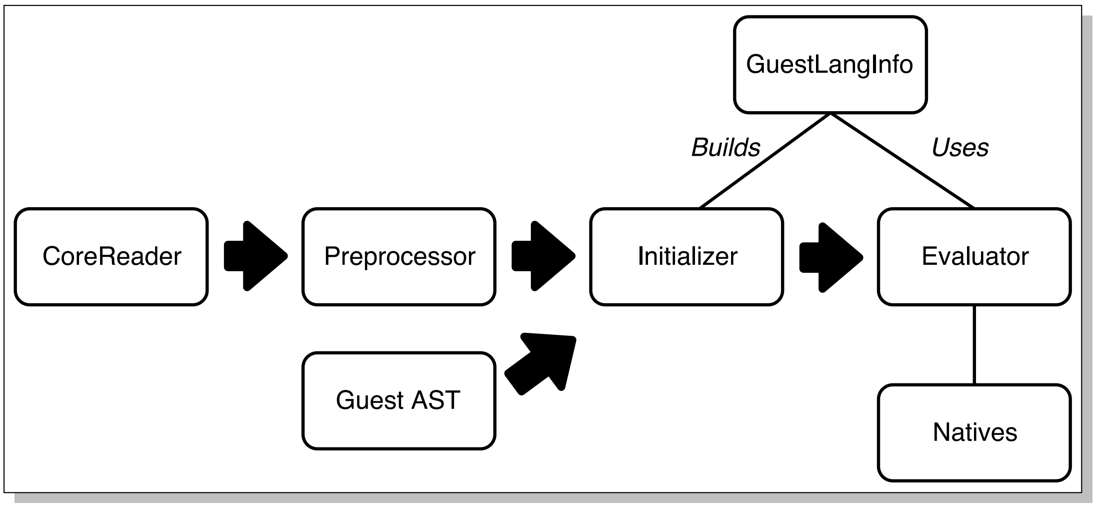


Figure 5.1: Architecture of the *DsRs* crate

Figure 5.1 shows an architecture for *DsRs*. First, the contents of a DynSem core specification are read. Then the preprocessors preprocesses this specification. It annotates native operators, interns constructors and creates typing information. Based on this annotated specification, the initializer builds the rule registry and transforms the guest language AST to an AST which is optimized for the DynSem interpreter. Based on this new AST, the language info and the natives mapping the evaluator executes the program.

5.2 Introduction to DynSem

DynSem models the semantics of a language as a set of reduction rules. During execution, the DynSem meta-interpreter recursively applies these rules to the terms of the abstract syntax tree starting at the top. To apply a rule, the interpreter starts by matching the current term against the left-hand side of the rule. After this match succeeds, it evaluates an optional set of premises. Finally, when all of these premises evaluate successfully, the interpreter builds the target pattern.

Figure 5.2 shows an example of a reduction rule which specifies the semantics of adding two integers. Assume we want to reduce the term `Add(Lit("1"), Lit("2"))`. During matching DynSem compares the names of the constructors, it binds `Lit("1")` to variable `e1` and `Lit("2")` to variable `e2`.

Evaluating the premises is the next step. In the addition example, the interpreter evaluates `e1 --> NumV(i1)`. First the term `e1` of this reduction will be build, this results in `Lit("1")`. This is the term the interpreter will try to reduce. Assuming this reduction succeeds and returns a `NumV(1)` term, this term will be matched against the right-hand side of the reduction `NumV(i1)`. After this match `i1` binds to `1`. Similarly, the second premise is evaluated. When this evaluation succeeds, the interpreter builds the right-hand side term of the rule: `NumV(addI(i1, i2))`. `addI`

```
rules

  Add(e1, e2) --> NumV(addI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2)
```

Figure 5.2: Semantics of Addition in DynSem

```
rules

  If(BoolV(b), ontrue, onfalse) --> v
  where
    case b of {
      true => ontrue --> v
      false => onfalse --> v
    }
```

Figure 5.3: Conditional Execution in DynSem

is a native operator which adds two integers together and returns this result, native operators are explained more deeply in section 5.4. The result of evaluating this rule is `NumV(3)`.

In the example, both premises are reduction premises. Such premise is a premise which builds some term at the left hand side and then tries to reduce this term. After the reduction returns, it matches the returned term to the right-hand side pattern of the reduction. DynSem has two other kinds of premises next to the reduction premise: match and case premises. Match and equality premises execute a match from one term to the other or compare equality, they succeed when the two terms match/are equal. For example, `n => Sub(e1, e2)` premise matches term `n` against pattern `Sub(e1, e2)`.

Case premises are similar to switch case statements found in other programming languages and enable conditional execution. A case premise build some term and then tries to match this term to a couple of other terms. When one of the terms matches, the premises associated with this term are executed. Sometimes none of the cases match, in this case the premises associated to otherwise statement are executed. An example of a case premise can be found in figure 5.3. Here, the interpreter matches `b` against `true` or `false` literals. When `b` is true, the interpreter reduces the `ontrue` term else the `onfalse` term. Note that `case` can also handle more elaborate matches.

5.3 Rule Lookup

During execution, it is necessary to quickly look up all rules applicable to the current term. A naive approach would be to iterate over all possible matches, which could potentially result in a traversal over all rules for each rule call. This traversal would have a negative influence on the performance of the interpreter. We reduce this footprint by decreasing the amount of potentially applicable rules.

To reduce the amount of potentially applicable rules, DsRs builds a rule registry. When a rule is loaded into this registry, it is determined what kind of match the left-hand side term of a rule has. Determining this match kind of a term should be fast. Therefore the runtime of the kind construction algorithm is bounded by the match pattern depth of the left-hand side of a rule, which in practice is low. We distinguish the following four kinds of left-hand side matches:

Constructor Default match on a constructor or metafunction. Based on the name of the constructor and the number of arguments.

Exact List Match Match on a list with an exact number of arguments.

Head Tail List Match Match on a list where at least a number of elements need to be present, but at the end of the list there is a tail element.

Other All matches which do not fall in all previous categories. For example Literal String matches. In practice this kind of matches are rare.

During the execution of a DynSem specification, whenever a rule call is done the interpreter constructs a term kind. It compares this kind to the kinds present in the registry. Based on this, the comparison function returns a set of potential matches. For lists, this set is a union of multiple match kinds. For example, a list match with length 3 can be matched against all head-tail matches where the head size is either 1, 2 or 3, and all matches which match against a list of exactly 3 items.

Theoretically, when a list contains 99 elements, it would be necessary to lookup all head tail matches where the head size would be 1 to 99. In practice, most matches include less than 5 matches on individual elements of a list. To set a bound on the number of lookups which need to be performed when the potential match set for a list is constructed, DsRs keeps track of the maximal number m of individual element matches in the specification. If a list is longer than this maximal match number, the interpreter returns the set which would be returned for a head tail match of length m. Results of rule lookups are cached and calculated lazily.

5.4 Natives

In some cases an external definition of the semantics of an operation is necessary. For example, the meaning of adding two integers together is something which should

```
signature
  sorts
    Exp
    V

  constructors
    Lit  : INT -> Exp
    Plus : Exp * Exp -> Exp
    NumV : INT -> V

  native operators
    nativePrint: V -> V
    newBigInt: String -> BigInteger

  native
    "java.math.BigInteger" as BigInteger {
      add : BigInteger -> BigInteger
      subtract : BigInteger -> BigInteger
      multiply : BigInteger -> BigInteger
    }
```

Figure 5.4: Constructor and Natives Declaration in DynSem

be defined external to the semantics of a language. To abstract away these implementation details, DynSem offers two language constructs: native operators and native data types.

A native operator is a function which is known to DynSem by only its signature. So the only thing DynSem knows about such operator is that it takes a number of arguments and returns some value of a particular type, the underlying implementation is a black box. In 5.2 `addI` is a native operator which takes two integers as input and returns the addition of these two integers. Typical examples of native operators next to integer arithmetic are string operations and standard io (as can be seen in figure 5.4).

DSJava implements native operators using runtime reflection. Currently, Rust lacks the abilities to do reflection, therefore during the initialization step, one of the arguments a user should provide is a list with a mapping from the name of an operator to the corresponding function. All native functions have the same signature and take two arguments: a vector of DSObjects and a service provider. Currently, this provider provides services such as parsing and desugaring of guest language programs. These services enable guest languages to add functionality like file importing or self-evaluation. Figure 5.5 shows the definition of a native operator in Rust. To simplify between DynSem and Rust, *DsRs* exposes functions for building constructors and converting Rust native types to DynSem types.

```
fn str_ends_with(
  mut args: Vec<Rc<DSObject>>,
  _service_provider: &mut NativeServiceProvider
) -> NativeResult
{
  let string : String = args.pop_front_as()?;
  let pattern : String = args.pop_front_as()?;

  Ok(string.ends_with(&pattern))
}
```

Figure 5.5: Native Operator implementation in Rust

Native data types are an extension to the type system of DynSem. DynSem knows a native data type as some object which implements an interface. The actual implementation of this interface is left to the host language of DynSem. In Figure 5.4, BigInteger is an example of a native data type. `newBigInt` is a native operator which returns a new BigInteger. DynSem and its type system know that this object implements three method: `add`, `substract` and `multiply` which take a self and another BigInteger value and perform some operation on them.

DSJava implements calling, constructing and converting native data types using runtime reflection. Therefore porting these types to Rust is a non-trivial task. *DsRs* only offers rudimentary support for Native Data Types by allowing a native to return a 'native object'. This is a wrapper around a usize, which a language developer could use as an index of a map which stores these objects. This map should be maintained by the language developer. When DsRs encounters a method call on a native object, it will call a special native operator. Making the type system of DsRs generic over some user-defined structure which implements a trait with a 'dispatch' method could improve this support in future versions.

5.5 Types

DynSem contains a static type system which is used during code generation to make some language features work. Porting these features to *DsRs* created some challenges and induced certain design decisions. This section gives a general introduction to the DynSem type system. Furthermore, it discusses the challenges and solutions which were necessary to get typing to work in Rust.

5.5.1 DynSem Types

DynSem its type system is static and consists of three parts: built-in types, constructors and sorts. Built-in types are types DynSem provides, these types fall in two categories simple and collection types. Simple types in DynSem are integers, floats, strings and booleans. They support matching and equality operations. DynSem has three types of collections: Maps, Lists and Tuples. All collection types are immutable, this means that operations on these collections create a new collection rather than mutating the original collection. Note that tuples have a fixed size and one cannot perform collection operations on it.

Constructors have a name and a number of statically typed children. Children can be any type or sort. For example in figure 5.4 `Plus` is a constructor with two children of type `Exp`. Each constructor has a sort, which is a type which describe multiple constructors. For example, in figure 5.4 `Exp` and `V` are sorts. Whenever the type `Exp` is used, it could be a `Lit` or a `Plus`.

For the simple types *DsRs* uses the default primitive types provided by Rust. Due to the immutable nature of collections such as HashMaps, the default implementations Rust provides are not suitable. For example, to keep the existing map from changing the whole map needs to be cloned when inserting an element into Rust its default HashMap. To keep a balance between cheap lookup and insertion *DsRs* uses Hashed Array Mapped Tries [6]. Languages such as Clojure use this structure to implement fast immutable maps without cloning. *DsRs* uses the implementation of the Im-Rs crate, which provides a number of immutable data structures. For lists, normal Rust vectors are used.

5.5.2 Constructors

Due to the lack of code generation, *DsRs* stores constructors as a string with a vector of children rather than a typed class representation. Matching two constructors results in a string comparison. This comparison is unnecessary as all valid constructor names are known at the start of the execution. To improve the speed of matching, during initialization, the names of these constructors are interned.

Interning is the concept that a string is stored at a location in an immutable fashion. Only references to a string are distributed throughout the program. Therefore, when a comparison is done, two 'strings' are equal when they point to the same location (referential equality). This results in the comparison of two integers instead of two strings. String interning adds a small overhead during the preprocessing step, but this is compensated by a speed up at runtime. Theoretically this interning step sets a limit on the maximum amount of distinct constructors (max unsigned integer size of the compilation target), in practice this limit is so big that this does not cause any problems.

5.5.3 Ownership of Values

Rust has a strict ownership model where non-primitive types have an owner and a lifetime associated to it. A value cannot outlive its owner without the transfer of ownership to a new owner. DynSem however has no way of knowing how long a value will live, which in the worst-case could be the execution time of the program. Satisfying Rust its borrow checker can be achieved in 3 different ways. A naive option would be clone an object whenever a transfer of ownership is required. A transfer of ownership happens when a DynSem value is bound or stored in a DynSem Map. Biggest drawback of this option is that large parts of the AST are cloned which, especially at the top of the reduction tree, could become expensive.

DynSem is an immutable language, so once created values cannot be changed. Rust allows unlimited passing of immutable references as long as these references are kept alive and the underlying structure has an owner. This concept is used in the second option: have some factory own all terms and only provide references to those terms. This method has two drawbacks: 1) The lifetime parameter of this factory will cascade throughout the code base, 2) Terms stored in this factory can only be safely destroyed when the program ends. This results in a big memory footprint when running a program in this interpreter.

DsRs uses Rust its *Rc* construct as a compromise between the previous two options. *Rc* is a structure which takes ownership of a value and keeps a reference count. When this counter reaches 0 the value is destroyed. Cloning a Rc creates a new Rc object and increases the reference count of the pointer. This operation is cheaper then cloning, but more expensive then passing immutable references as increasing the reference count introduces a runtime overhead. However, the decreased memory footprint outweighs this overhead.

5.5.4 Implicits

To keep a language specification concise, DynSem offers a number of language features. One of these features are implicit constructors. An implicit constructor is a constructor with a single child, for example `IntExp: INT -> Exp`. When an `Exp` is necessary but an `INT`, for example `10`, is found the interpreter automatically constructs a term `IntExp(10)`. This decreases the amount of manual conversion necessary in the semantic specification.

To implement implicits the existing DynSem interpreter relies on the interpreter generator. During code generation, this generator adds a fallback option to the term factory classes. *DsRs* skips this code generation step and has therefore no generated factory classes. This made a direct port to rust not an option. To add support for implicits to *DsRs* we developed another solution.

Implicit conversion is done during the term build. In the preprocessing step the preprocessor annotates all locations where potentially an implicit build is possible. At these locations a type check is performed at runtime to test whether an implicit conversion could be necessary.

To determine which implicit conversion between two types are possible, the preprocessor constructs an implicit graph. Sorts are represented as vertices, when a sort S_1 is implicitly convertible to sort S_2 using constructor C a new edge is created between the node of S_1 and S_2 with value C_1. When two sorts S_1 and S_2 are encountered, an implicit can be build by finding a path between S_1 and S_2 and use all implicit constructors on the edges of this path.

```
rules

  Box(v) --> BoxV(allocate(v))


  allocate(v) :: H --> RefV(i) :: H {i |--> v, H}
  where
    fresh => i

  E | bind(id, e) --> {id |--> v, E}
  where
    e --> v
```

Figure 5.6: Example of Semantic Components in DynSem

5.6 Context

Using just reduction rules a language designer is able to concisely specify a stateless language. In practice most programming languages offer some state in the form of a store and an environment. Introducing state in the terms could be done using just reduction rules and terms. This becomes verbose pretty fast. To ease the passing around of state DynSem has the notion of semantic components. An example of components in Dynsem can be found in figure 5.6.

DynSem has two types of components: read-only and read-write components. In figure 5.6 `H`, which represents a storage, is an example of a read-write components. DynSem passes these type of components both horizontally and vertically in the reduction tree. Component `E`, which represents the environment is an example of a read-only component. These components are only passed down implicitly.

By default Dynsem passes semantic components implicitly. Only when a rule uses a component it should explicitly match and pass the components. This implicitness allows the specifications of DynSem to be more concise. Although the specification the rule in figure 5.2 belongs to could contain components, the rule can omit these as it does not used any of these components. In figure 5.6 the `Box` rule omits the mention of the `H` and `E` components as it does not use these components.

During interpretation, rules do not share state other than the current term and the semantic components. Rules in *DsRs* stores the current term, local variables and components in a frame. When a rule calls another rule, it initializes the new frame with the to be reduced term and sets the semantic components it wants to pass down. The new rule matches these semantic components. After execution, the right hand side of a rule sets the components the rule wants to pass up and changes the current term to the term that was build. Using this information the caller matches the returned component and terms, then it discards the frame.

```
rules

  e@While(cond, e2) --> vr
  where
    cond --> BoolV(b)
    case b of {
      true => e --> vr
      false => vr => UnitV()
    }
```

Figure 5.7: Semantics of a While loop in DynSem

Next to state related to rules, *DsRs* also maintains state about the guest language during interpretation. This is immutable state with information about the guest language and the specification which is being interpreted. Information about rules, types and natives are kept in this state. Furthermore, it provides references to the parser and desugarer which are passed down the tree to enable native functions to parse and desugar files.

5.7 Interpretation

Dynsem is a set of reduction rules which are applied recursively. The recursive implementation of the original meta-interpreter reflects this. When expressing the semantics of iteration this recursion becomes a problem as DynSem has no native support for iteration. Therefore, recursive application of a reduction rule is the only way of supporting iteration. Semantics for languages including iteration would contain a reduction rule in a form similar to figure 5.7.

In this code variable `e` binds to the `While` term, and the condition of the while loop is evaluated. When the condition is false, the loop will stop and return to the previous rule, but when the condition evaluates to true, the interpreter will recursively call the same rule. In the Java back-end, this results in a new recursive method call at each iteration. The maximum amount of iteration in a program is bound by the stack size of the JVM, this bound could potentially result in a stack overflows due to iteration.

Although increasing the stack size 'fixes' this problem, it merely addresses the symptom. The actual problem is the lack of tail call elimination in DynSem. Originally, the Java back-end had some form of tail call elimination, but in some cases, this gave incorrect results. One possibility would be to detect tail calls by doing static analysis, this requires the rules to be in some specific forms, and it does not work in the case where recursion is indirect.

To solve the overflowing stack problem we approach this at an interpreter level. If the interpreter could be written as a while loop, where every recursive method call is bound to some low number, the stack depth could be decreased. Optimally, such interpreter would function within the realm of safe Rust and would require no changes to the DynSem language.

One option would be to introduce a native operator who executes the while loop. However, the current semantics of native operators is that native operators cannot return semantic components or rule results. Therefore the use of such native operator would be awkward from a user perspective as such operator would not preserve state. Furthermore, this option would add extra native calls to the code base which decreases the readability and conciseness of DynSem.

As a solution, we develop an evaluation algorithm which serializes the application of a reduction rule into a number of higher-level operations called continuations. This algorithm stores the continuations on a stack, which is stored on the heap rather than the actual stack. At every step, the interpreter pops the top element of the stack, evaluates it, and pushes the potential new continuations on the continuation stack. The following four types of continuations are distinguished:

- **Evaluate Premise** continuation evaluates a premise. During the evaluation of a reduction premise it builds the left-hand side, looks up the serialized version of the corresponding rule and returns the continuations.

- **Build & Return** builds the right-hand side and the components which should be returned. It stores the variables as described in section 5.6.

- **Match Returned** matches the components returned by the previous rule to the left-hand side of the originating reduction premise. It also matches all returned components.

- **Other Rules** During a normal evaluation this continuation returns nothing. If the interpreter encounters an error and unwinds the stack this continuations stops the unwinding when it has an alternative rule which the interpreter can apply.

- **Match Input** Matches the input of a rule. Succeeds when the inputs matches.

For example, the algorithm serializes the rule in figure 5.7 into the following continuations: Match Input, Evaluate Premises and Build & Return. These continuations are pushed onto the stack in the reversed order during the initialization step.

Sub-algorithms 8, 9, 10 and 11 give a global overview of the evaluation algorithm. This algorithm keeps track of two stacks: the continuation and frame stack. A frame is the context of a rule, so only when a new rule is called a new frame is pushed onto the stack. Continuations of the type 'match and return', discard a frame after it is matched against the right-hand side of a rule. For simplicity, in this algorithm the frames stack is a global, but in the implementation it could be passed around.

Algorithm 8 Iterative Evaluation Algorithm

```
 1: $frames \leftarrow []$
 2: function EVAL($currentTerm$)
 3:     $stack \leftarrow [continuations\ initialisation\ rule]$
 4:     $error \leftarrow \emptyset$
 5:     $frames \leftarrow$ [newFrame(currentTerm)]
 6:     while !$stack$.isEmpty do
 7:         $continuation \leftarrow stack$.pop()
 8:         if $error = \emptyset$ then
 9:             $result \leftarrow$ execute($continuation$)
10:             if $result$.isError then
11:                 $error \leftarrow result$.error
12:             else
13:                 $stack$.push($result$.new_continuations)
14:             end if
15:         else
16:             $alternatives \leftarrow$ unwind($continuation$)
17:             if !$alternatives$.isEmpty() then
18:                 $stack$.push(alternatives)
19:                 $error \leftarrow \emptyset$
20:             end if
21:         end if
22:     end while
23:
24:     if $error \neq \emptyset$ then
25:         return $error$
26:     else
27:         return $frames$.pop().term
28:     end if
29: end function
```

Algorithm 9 Call a Rule

```
1: function CALL_RULE($inputTerm, ruleName$)
2:     $ruleset \leftarrow$ lookup(inputTerm, ruleName)
3:     $continuation \leftarrow$ ruleset.pop()
4:
5:     $frames$.push(newFrame($inputTerm$))
6:
7:     return [OtherRules($ruleset$)] ++ continuations
8: end function
```

During the first step, the iterative evaluation algorithm initializes the continuation stack with the continuations belonging to the initialization rule. Then it initializes the frame stack with a frame based on the outer term of the AST. After initialization finishes, the evaluation loop begins. This loop continues until the stack is empty. Finally, when the stack is empty and no error is active the evaluation succeeds and the eval function returns the value the initialization rule returned.

Algorithm 10 Match Returned Term

```
1: function MATCH_RETURNED(rhs)
2:     match(rhs, frames.pop())
3:
4:     return []
5: end function
```

Algorithm 11 Unwind Stack

```
1: function UNWIND(continuation)
2:     switch s do
3:         case MatchReturned
4:             frames.pop()
5:         case OtherRules(set)
6:             new ← set.pop()
7:             frames.peek().empty()
8:             return [OtherRules(set)] ++ new
9:         case _
10:
       return []
11: end function
```

In the evaluation loop, at each iteration the algorithm pops the top continuation of the stack. If no error is currently active, it executes the continuation. A continuation returns either a (possible empty) set of new continuations or an error. New continuations are pushed onto the stack, when an error occurs the stack starts unwinding. During unwinding the algorithm empties the stack until a 'other rules' continuation is found which returns an alternative rule which could be applied. Note that in this case 'match returned' continuations will still discard the top frame, but they don't execute the match.

When an 'evaluate premise' continuation encounters a rule call, the 'evaluate premise' continuation looks up the set of potential rules based on the input term and the name of the rule. It creates a new frame and returns the continuations of the first potential rules prepended by the 'other rules' continuation. During the return phase, the 'build and return' continuation leaves the frame in the state described in section 5.6. Then the 'match and return' continuation matches the current frame

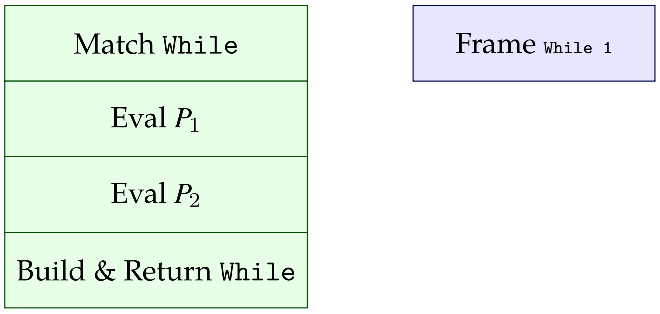


Figure 5.8: Continuation Stack (left) and frame stack (right) after initialization

to the right hand side of the reduction premise. Finally, it discards the frame and returns an empty set of continuations.

In figure 5.7, the code would initialize the stack like figure 5.8 on the input of term `While(Cond, Stmts)`. In this term `Cond` is some arbitrary expression which reduces to a boolean and `Stmts` is a list of statements which get executed. First, the interpreter matches the left-hand side of the rule.

After the match succeeds, *DsRs* splits the rest of the rule into two continuations: an 'evaluate premise' and a 'build & return' continuation. These continuations are pushed on top of the stack so that the continuation which gets evaluated first is on top. The interpreter will pop the top premise of the stack and evaluate the case statement. Assuming the boolean evaluates to true this will result in the `e --> vr` premise to be pushed on the stack, which results in a recursive rule call.

During the evaluation of this recursive call, the premise builds the left-hand side of the rule and sets it as the current term. Then it will push a 'match returned' continuation on the stack. Just like the previous rule, the left hand side gets matched. Lets assume that in this case the condition evaluates to false. After this step the continuation for the evaluation of match premise `vr => UnitV()` gets pushed onto the stack, which evaluates without adding new continuations to the stack. With no new continuations pushed onto the stack, the 'build and return' continuation gets executed. This continuation builds the term `vr`, this term is returned to the previous rule call and matched against the `vr` term in the right hand side of the `e --> vr` premise. This binds `vr` to `UnitV()` which is the value that will be returned when the right-hand side of the rule is build.

5.8 Evaluation

DsRs is a port of an existing back-end, to get an idea how it compares to the existing back-end we do an evaluation. This section presents the results of this evaluation. First, it presents the results of the validation and then shows the results of the performance comparisons.

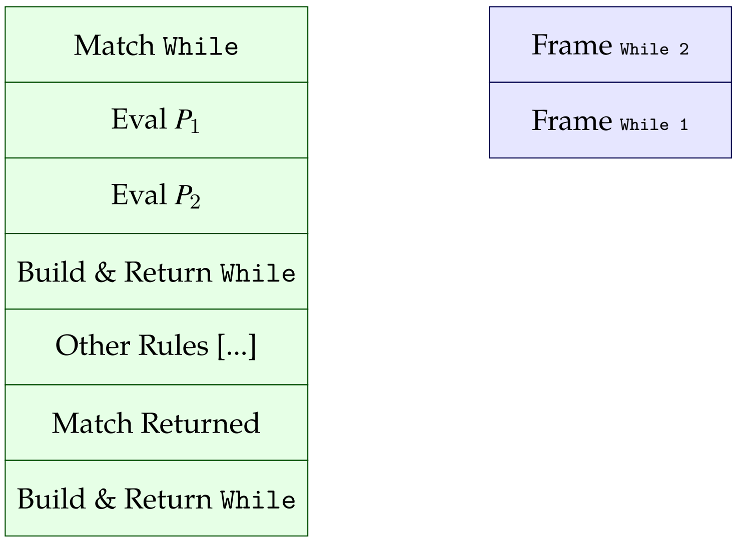


Figure 5.9: Continuation Stack (left) and frame stack (right) after new rule call

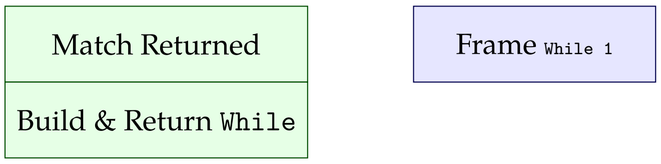


Figure 5.10: Continuation Stack (left) and frame stack (right) after the recursive call returned

5.8.1 Validation

While porting, mistakes may lead to differences between the existing back-end and *DsRs*. To ensure that there are no significant semantic differences or missing features, we executed a validation of *DsRs* by running the test sets for SL, Tiger & Grace. These specifications are the largest DynSem specifications currently available. Both SL and Grace have a test set available which consists of: test programs, files with the input to these programs, and the output expected on the input of these programs. For Tiger, a similar test set exists, but the output and input files were not available. Therefore, these files were created manually for a subset of the test set.

To automate the testing we developed a test framework which reads this format. It overrides the input and output native operators of the guest language and attaches these to a test structure which keeps track of the expected input and output. When a test encounters a difference in the asked input or the given output of a program it fails. Note that these tests succeed or fail based on IO, they cannot detect different execution paths which give the same IO.

	SL	**Tiger**	**Grace**
Total Tests	33	20	254
JVM Pass	33	20	240
Rust Pass	33	19	240

Table 5.1: Results of Validation

Table 5.1 shows the results of this validation. *DsRs* passes the same amount of tests as the Java implementation for the Grace specification. Due to some missing feature of the parser one tests which work on Java fails in the Rust implementation in the Tiger test set. This failing test works at an AST level, so this is more a parser issue than an interpreter issue. For SL, both interpreters pass the same amount of tests

5.8.2 Performance

Figure 5.11 shows the results for the SL benchmark. As expected, the manual implementation of SL is the fastest performing interpreter, followed by the vanilla Java back-end of DynSem. Graal is slower in this benchmark, probably because Graal invests some overhead during execution which for short running programs it cannot regain. When *DsRs* is run natively it performs reasonably fast considering it is a full interpreter for the DynSem language. Running the same code on WebAssembly induces a 6-8 times performance hit.

Results for the Tiger benchmark can be found in figure 5.12. Due to some exceptions which were only encountered when running the benchmarks on Graal, the Graal platform was removed from this comparison. Notable in these results is that the performance penalty WebAssembly introduces is much lower than in the SL benchmark. It is unclear why this is the case, the native implementation as expected relative to the Java implementation. WebAssembly is the outlier in this case.

The graph in figure 5.13 shows the results for the Grace benchmark. Compared to the other two benchmarks, Grace programs, in general, are much slower. Runtime typing and property access checks on objects are the cause of this slowness. Furthermore, the Grace specification contains more semantic components and other complexity compared to the other two languages. This benchmark does not include the Array tests as there are no reasonably fast array/list implementations in Grace.

Depending on the languages WebAssembly introduces a 2-8 times performance hit. Is this the expected performance penalty? Herrera et al. benchmarked a set of C programs compiled to WebAssembly in different browsers [20], they found depending on the program a 2 times performance penalty. Our results differ from the results obtained by Herrera et al. We have multiple hypotheses why this is the case.

One option is that this difference depends on the executed program. In the C benchmarks a difference between programs was found, some programs were even faster than their natively ran counterpart. Another options is that it depends on the

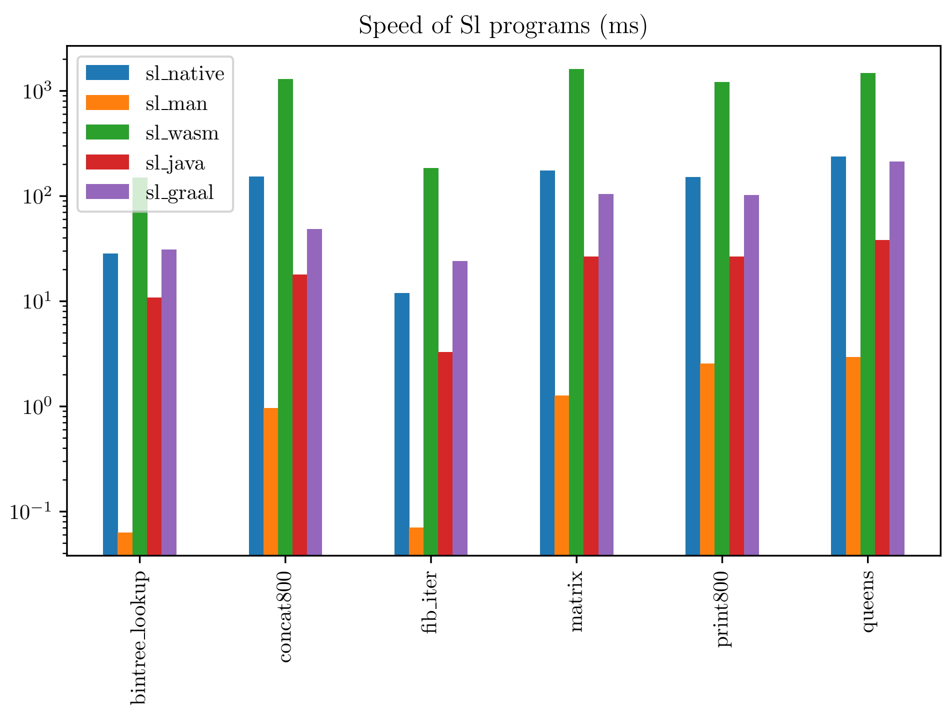


Figure 5.11: Benchmark Results for SL language (*sl_man* is the manual SL port ran natively)

different compilers, the C benchmarks were compiled using Emscripten, our benchmarks use the younger official LLVM back-end. Lastly the amount of Javascript interoperability might influence the benchmarks. Where our benchmarks have a relative high amount of output which requires JS interop, the C benchmarks print only a few lines.

In general, the *DsRs* implementation of DynSem is slower than its Java counterpart. When *DsRs* is compiled to the browser this slowness increases. To some extent, this slowness can be explained. First of all the Rust implementation has a general algorithmic disadvantage as it behaves more like an actual interpreter and uses no generated code, this induces a performance hit as it removes information for the Rust compiler. Secondly, the JVM can do optimization which cannot be done at compile time due to the JIT nature of the JVM. JMH let Java do all these optimizations before actually reporting the time it took to execute the program, cold runs of a program are in general 3-4 times slower.

To improve the performance of the Rust implementation a few possibilities exist. Although *DSRs* its rule dispatch algorithm already reduces the number of possible rules, it still has to try multiple rules in the worst case scenario. Compiling pattern matches to decision tress could lead to the execution of fewer operations during rule dispatching. [8]

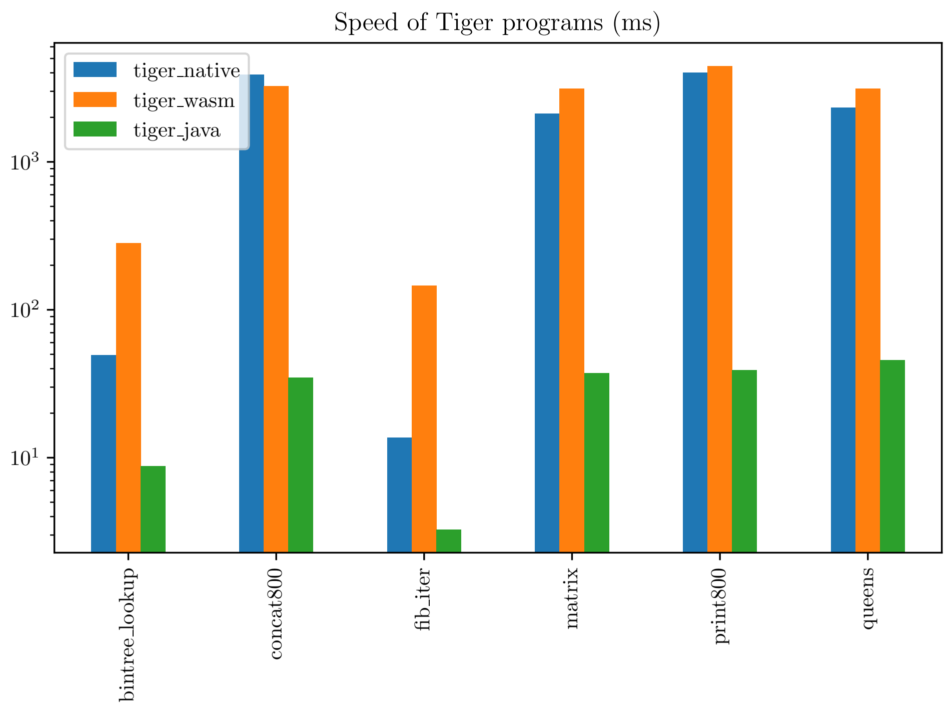


Figure 5.12: Benchmark Results for Tiger language

A second option which would require some bigger changes is the addition of runtime specialization to the Rust interpreter. For this to work correctly it would require more knowledge about the whether the provided natives are pure. If this knowledge is not provided the meta-interpreter cannot perform optimizations on all reductions which involve natives as they are a black box.

Lastly, adding a Rust code generation back-end to the existing DynSem back-end could be an option. The generated code would improve the knowledge the Rust compiler has about the terms and rules. Matching in some cases could be replaced by dispatching certain functions and using factories implicit conversion become cheaper.

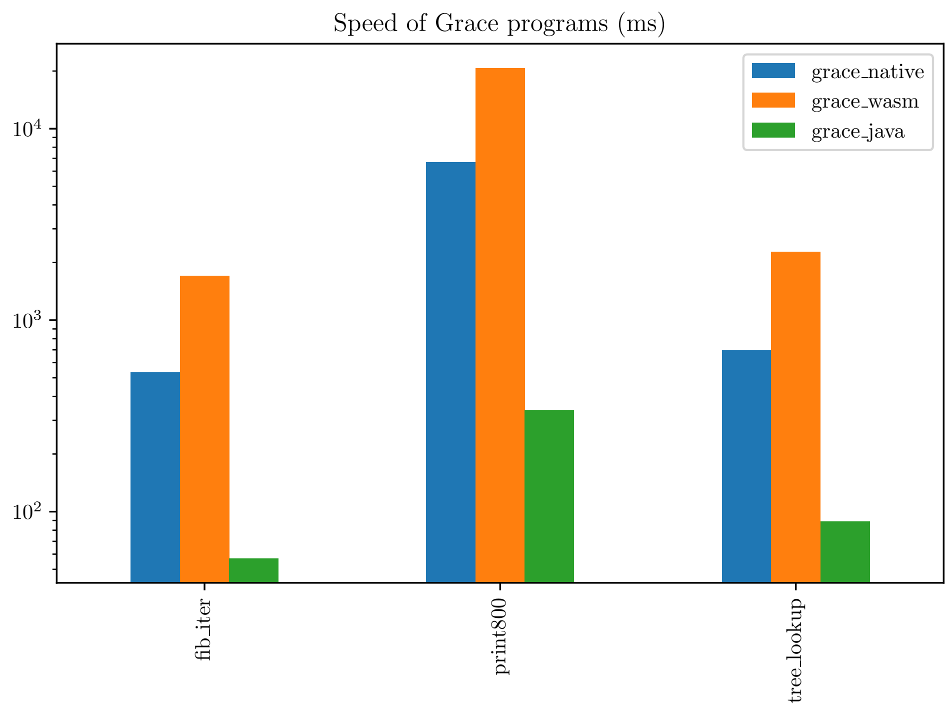


Figure 5.13: Benchmark Results for Grace language

Chapter 6

Related Work

Currently certain technologies exist to edit and run code through a web browser. Many of these technologies however are not language parametric nor client-side. In this section we will explore these existing technologies.

6.1 Web-Based Coding Environments

Web-based code editors fall apart in two categories: playgrounds and full-fledged web-based IDEs. Many programming languages offer some form of an online playground, which often executes their code on a server. Examples of language playgrounds are Rust Playground [38], Go Playground [45], Scastie for Scala [39] and JsFiddle [25]. From these environments, JSFiddle is the only one who executes the code written by the user in the client. For this, it uses the eval function of the JavaScript interpreter provided by the browser. All these playgrounds are tailored to the language they execute and are not generic.

In both academic and commercial environments, web-based IDEs were developed over the years. Cloud9 [5] is the leading commercial web IDE. Using Ace editor as a basis, it supports all languages Ace supports for syntax highlighting. Highlighting in ace is done using regular expressions, no actual parsing is done client-side. Typing and more advanced editor services run in a hybrid server-client model. Cloud9 executes programs server-side on a Linux virtual machine.

Eclipse Che is an open source web IDE [14]. For editing, it uses the Orion Editor. Che consists of a thin front-end which communicates with the Che Server, this server provides all services, plugins, and files. [13] Programs are executed on the web server. Che is used as a basis for multiple commercial Web IDEs. Arvue [3] is an academic Web IDE developed by Aho et al. It allows the in-browser development of web applications and enables developers to easily deploy these languages to the cloud. Codiad [9] is an open source Web IDE with the focus on simplicity. Execution of programs is done on a server.

All these Web IDEs execute syntax highlighting on the client-side. Other services run partly on the client and the server. Existing plug-ins for languages cannot be reused. For each new language, a plug-in specifically made for that language should be created. Every mentioned IDE executes its code server-side.

6.2 Language Portability

Bringing languages to a Web IDE is an example of the IDE portability problem. This problem is that for m languages and n IDEs m+n implementations are necessary to run every language on every IDE. Monto [42] tries to solve this problem by decoupling the components of an IDE. Central in its design is the message broker which communicates between different language implementations and the editor. Due to the message broker, a language only has one single instance to communicate with for multiple IDEs. To introduce caching Monto requires services to be stateless. This requirement disallows the introduction of stateful services. For performance reasons algorithms used in a workbench could be stateful and therefore hooking Monto to a language workbench could become a problem.

Language Server Protocol [30] solves the portability problem similarly by introducing a language server which communicates through a certain interface with the language server. Problem with this approach is that LSP communicates to a server, this could introduce a delay in editing. Moreover, syntax highlighting and parsing should be done client-side, which requires client-side parsing.

Often languages have some back-end to target the browser. Both Scala [11] and the Akka toolkit [43] have a target in written in JavaScript. Go has a GopherJs [2] which is a compiler which takes Go as input and compiles it to JavaScript. Google Web Toolkit is a set of tools which allows developers to develop web applications in Java. Applications can be executed on the JVM for debugging purposes or can be compiled to JavaScript and HTML to use in the browser. [17] The Kotlin language can also be compiled to JavaScript. [29] Emscripten [55] is an LLVM to JavaScript/WebAssembly compiler which also simulates certain syscalls in the browser. Python has a source to source Python to JavaScript mapper Jiphy. [22] MiniGrace is a Grace compiler which can compile a grace program to JavaScript and C. [41] These backends are not generic and are often a separate project. Due to browser limitation they often lack certain features which are included in the original compiler or interpreter.

6.3 Language Development Tools on the Web

6.3.1 Language Workbenches on the Web

In this work artifact generated by the Spoofax Language Workbench are used. Spoofax is not the only workbench. Xtext is a set of programming languages and DSLs. It can generate a web editor which uses the Language Server Protocol to deliver type checking and code completion in the browser. Besides this, it has no support for bringing languages to the browser. [15] MPS is a workbench developed by JetBrains, it has support to generate JavaScript code based on an existing program.

[34] Rascal is programming language which integrates certain parts of the design of a programming language into its language primitives. [27] It has no support for web-based IDEs.

6.3.2 Parser Generators on the Web

SDF and the associated SGLR algorithm [48] belong to the family of parser generator framework. Multiple other formalisms in this domain exist. ANTLR [33] is a formalism widely used to define grammars. Based on a language definition ANTLR generates an LL(*) parser. ANTLR has different code generating back-ends targeting Java, C++, C# and Python and JavaScript. Using the JavaScript back-end ANTLR can be brought to the web.

Yacc [24] and Lex [32] are often used together to write parsers. Yacc is a parser generator and Lex is able to generate a Lexical analyzer. Bison [12] is Yacc compatible parser generator, the original implementation does not support web targets. Jison [23] is a Bison compatible parser generator which serves as a JavaScript back-end for Bison.

Parsing Expression Grammars are a syntax formalism and parser generator developed by Ford.[16] This formalism is similar to the notation of Context-free Grammars. When a PEG parser finds an ambiguity, it selects the first option. Over the years PEG-based generators were developed for multiple languages. Peg.js [35] and Peg.js-fn [36] are both able to generate JavaScript based parsers which can be used on the web.

SGLR is a parsing algorithm which is able to recognize more types of languages than ANTLR, Yacc and PEG. Currently no fast SGLR implementations exist in the browser. Therefore, language developers can parse more languages in the browser using *SGLRs*.

Chapter 7

Future Work

Spfx_rs is a prototypical Spoofax back-end which evtuallyentually should lead to full Web IDE integration of Spoofax. This chapter provides a high-level overview of this process and how it should be done. Furthermore, it discusses some improvements which could be made to *spfx_rs*.

7.1 Language Workbenches in the Browser

One of Spoofax its main goals is to make language development more pleasant. Currently, using Spoofax in the context of Web IDEs is not a pleasant experience. In the future, this experience should become just as pleasant as deploying languages in a classic IDE. *spfx_rs* is one step towards this goal. In this section, we give an overview of what the next problems and the next steps are towards this goal.

spfx_rs currently consists of a parsing, transforming and execution environment. Most of these services do not offer all features their Java counterparts offer. Furthermore, services such as NaBl, Spoofax its name binding and typing DSL are missing from the pipeline. These services should be included in future versions.

Now that (parts of) programming languages are able to run in the browser it should be more feasible to integrate Spoofax based languages in a web IDE. This integration could introduce some problems as not all algorithms implemented in *spfx_rs* operate in a multi-file environment where knowledge about the files in a project is distributed. Algorithms and packages language workbenches use often assume fast and total access to all data within a project. Clients of web-based IDEs however only have knowledge about the files which are edited, this results in either having full knowledge or quick response times.

When an editor service requires both full knowledge and quick response times this introduces a problem. To resolve this problem, multiple approaches could be tried. One could be to bring ownership of all files and services to the client and use the server only as storage. For big projects, however, this could become CPU and data intensive. Furthermore, it would require a port of all services to the client, which is non-trivial for all services.

Running everything entirely server-side however is also not the perfect solution. Many services offered by an IDE need a quick response time to feel responsive. But the roundtrip time to a web server might be too slow and give problems. In the end, a hybrid solution is probably the best alternative. To this end, existing algorithms should be changed so they can run in these situations.

For Spoofax, the best idea would be to get basic Web IDE support to work by supporting the Language Server Protocol for the services this protocol supports. Then over time, Spoofax could gradually move services from the server to the client. Syntax highlighting and parsing are good first candidates. Some services might need a hybrid server-client solution to both get a quick response and have full knowledge of a project.

7.2 DynSem

DynSem Standard Library Native Operators are an important way of abstracting away some implementation details in a specification. Currently, all DynSem specifications implement the same set of operators (e.g., string operations, integer arithmetic) which leads to unnecessary code duplication and more manual porting. Adding a standard library which includes these kinds of operators would increase the usability of DynSem.

Debugging Debugging a DynSem specification is, from a usability perspective, not a pleasant experience. Currently, DynSem provides two tools for debugging a DynSem specification. First of all debugging based on printing lines to the standard output, which clutters the specification with superfluous native calls. Another option is leveraging the debugging capabilities of the host language. This option involves setting the right breakpoints in generated code. Having the ability to set a breakpoint in DynSem could improve the experience of using DynSem. For this to work, DynSem core files should preserve origin information.

Code Generation Currently *DsRs* skips the code generation step of the DynSem interpreter generation process. Skipping this step results in a performance overhead. For small, short running, not performance sensitive programs which execute in a language playground this is not a big problem. For larger programming languages these performance implications might become problematic. Adding a code generation step would probably reduce the difference between the Java and Rust version of DynSem.

7.3 Spoofax in Rust

Better Native API Currently the native API works fine for regular native operators. However, *DsRs* implements native data types in a user-unfriendly way. Exposing a better API for this would improve the usability of Spoofax in Rust from a developer's perspective. Adding the ability to make the DynSem type system in Rust generic over some user-defined structure which implements some dispatch trait would improve both the speed and usability of Native Data Types.

Parser Improvements Spoofax in Rust implements a basic version of the SGLR algorithm. When the parser is not able to continue parsing it will report the (position of) last character it was able to reach. In practice, SDF3 provides some extra features such as auto-completion, error recovery and error reporting on a line based level. Adding these features to the current implementation would improve the ergonomics of the parser.

Origin Tracking and Runtime Error Reporting *DsRs* has some possibilities to do error reporting. Sadly the errors reported are not that great due to the lack of origin tracking in DSCore. It would be nice to be able to generate better stack traces with origin information.

Precomputation Using build.rs it is possible to generated code at compile time. By leveraging this code generation step, it might be possible to precompute structures of artifacts like the parse tables and rule registry. This precomputation would remove redundant information, which results in a smaller binary. Furthermore, this precomputation would speed up the runtime initialization step.

Web Application Integration Currently, the new back-end is only used in a language playground. Although this is a good use case, the new back-end also simplifies the integration of Spoofax language other than web-based IDEs. For example, DSLs for accounting, configuration or data model description which are developed in Spoofax could be used to partially evaluate certain expressions without adding extra load to the web server.

Scaling Currently the semantics of the resulting runtime are not fast enough. Using code generation and runtime specialization the meta-interpreter might get a performance more similar to their Java counterpart. This speedup would improve the automatic port for the current languages. It would be interesting to see how automatic porting scales for bigger languages when this new faster approach would be used.

Chapter 8

Conclusion

With *spfx_rs* developers can automatically port large parts of a language to the browser. For parsing and transformation, no manual intervention is necessary for the port and the resulting components have a performance which is similar to the Java back-end. Automatically porting the semantics of a language to the browser could be achieved to some extent. For some parts of the semantics, however, manual porting of code is still required.

The lack of a standard library in DynSem is one of the reasons why manual intervention is still necessary. Having no standard library results in the manual port of many simple operations which are implemented similarly in many languages. In practice, however, no standard library would entirely remove the need for natives. Only when natives would be written in a language which can run both on the JVM and the browser a fully automatic port would lie in the realm of possibilities.

What is the cost of automatically porting a language? For a language developer the costs of porting a language decrease with the creation of *spfx_rs*. Instead of porting the entire back-end of a language a developer is only porting the bits which are unique for its language. Although the cost of porting decreases, automatic porting introduces two extra costs.

First of all, an automatic port adds a performance penalty. For parsing, this penalty only consists of running in WebAssembly. However, skipping the code generation step for the meta-interpreter introduces an algorithmic cost. This cost, combined with the penalty of running WebAssembly results in the web back-end of the meta-interpreter being slower than the Java/Graal back-end.

Next to the performance penalty, automatic porting introduces a maintenance cost for the developers of Spoofax. Unless they decide to drop the Java implementation and only support the Rust back-end, they would need to maintain two back-ends. Both options require extra engineering work, so by offering automatic porting we transfer cost from the language developer to the Spoofax developer.

Given the same program, WebAssembly performs 2-8 times slower than the native platform. For parsing the resulting speed is fast enough. Compared to the old JavaScript versions of Spoofax libraries, this new implementation has performance characteristics more similar to the original Spoofax parser. These characteristics make the new parser more suitable to integrate it in web based development environments.

Compiling the Grace specification to WebAssembly results in a binary of 8.4 megabytes, only 250 kilobytes of this is Rust code. Spoofax compiled to Javascript had a runtime footprint of 420 kilobytes. Next to the runtime, the WebAssembly binary also contains the artifacts which SDF generates. These are 5.4 megabytes of .ctree libraries, 800 kb of DynSem core files and a parse table of 1.9 megabytes. Loading and initializing the Grace runtime takes 1.5 seconds on average, this is equivalent to loading Facebook or a big news site. To reduce the amount of data transferred in the future, research should be done on more efficient ways of storing and transferring Spoofax artifacts.

WebAssembly is a suitable compilation target for Spoofax based languages. Although WebAssembly has a performance overhead, this overhead is fixed. Furthermore, the binary size is equivalent or lower than Spoofax compiled to Js and has no parsing overhead. Note that these results are obtained using the current implementations of WebAssembly which are still in their early stages. Therefore these results could change in the future.

Overall our main conclusion is that creating automatic ports of languages defined using a language workbench, even for more extensive languages like Grace, is feasible and that WebAssembly is a suitable compilation target for this. Future work should be done on making the executable semantics in the browser as fast as their Java counterparts.

8.1 Discussion

Currently, *spfx_rs* is just a prototype. It would be possible to actually use it, but especially the lack of error recovery and syntax highlighting in the parser would impact the user experience in a negative way. Outside the realm of a language playground, the automatically ported execution environment is not fast enough. Adding a code generation step to *DsRs* would probably improve the speed of the meta-interpreter.

This work has two main contributions: First of all, we show that fast client-side SGLR parsing is feasible. For Spoofax this is important for integration with web-based IDE. Furthermore, we explored the idea of making all languages defined using SDF, Stratego and DynSem more portable by changing the back-ends of these meta-languages rather than a manual port.

Bibliography

[1] *A Cargo subcommand for the client-side Web*. URL: https://github.com/koute/cargo-web.

[2] *A compiler from Go to JavaScript for running Go code in a browser*. URL: https://github.com/gopherjs/gopherjs (visited on 03/26/2018).

[3] Timo Aho et al. “Designing IDE as a service”. In: *Communications of Cloud Software* 1.1 (2011).

[4] W Appel Andrew and P Jens. *Modern compiler implementation in Java*. 2002.

[5] *AWS Cloud9 Amazon Web Services*. URL: https://aws.amazon.com/cloud9 (visited on 03/26/2018).

[6] Phil Bagwell. *Ideal hash trees*. Tech. rep. 2001.

[7] Andrew P Black et al. “Grace: the absence of (inessential) difficulty”. In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM. 2012, pp. 85–98.

[8] Luca Cardelli. “Compiling a functional language”. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM. 1984, pp. 208–217.

[9] *Codiad*. URL: http://codiad.com/ (visited on 03/26/2018).

[10] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35 (2000), pp. 26–36.

[11] Sébastien Doeraene. *Scala. js: Type-directed interoperability with dynamically typed languages*. Tech. rep. 2013.

[12] Charles Donnelly and Richard Stallman. “Bison. The YACC-compatible parser generator”. In: (2000).

[13] *Eclipse Che | Extend*. URL: https://www.eclipse.org/che/extend/ (visited on 04/26/2018).

[14] *Eclipse Next-Generation IDE*. URL: https://www.eclipse.org/che/ (visited on 03/26/2018).

[15] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2010, pp. 307–309.

[16] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 111–122.

[17] *GWT*. URL: http://www.gwtproject.org/ (visited on 03/26/2018).

[18] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2017, pp. 185–200.

[19] MA Haisma. "Grace in Spoofax". MA thesis. the Netherlands: Delft University of Technology, 2017.

[20] David Herrera et al. *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices*. Tech. rep. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University, 2018.

[21] G Hoare. "The Rust programming language". In: *URL http://www.rust-lang.org* (2013).

[22] *Jiphy*. URL: https://github.com/timothycrosley/jiphy (visited on 03/26/2018).

[23] *Jison*. URL: jison.org (visited on 04/18/2018).

[24] Stephen C Johnson. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[25] *JS Fiddle*. URL: https://jsfiddle.net/ (visited on 03/26/2018).

[26] Lennart CL Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *ACM sigplan notices*. Vol. 45. 10. ACM. 2010, pp. 444–463.

[27] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. "Rascal: A domain specific language for source code analysis and manipulation". In: *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE. 2009, pp. 168–177.

[28] Donald E Knuth. "On the translation of languages from left to right". In: *Information and control* 8.6 (1965), pp. 607–639.

[29] *Kotlin to JavaScript*. URL: https://kotlinlang.org/docs/tutorials/javascript/kotlin-to-javascript/kotlin-to-javascript.html (visited on 04/16/2018).

[30] *Language Server Protocol*. URL: https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md.

[31] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[32] Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*. 1975.

[33] Terence J. Parr and Russell W. Quong. “ANTLR: A predicated-LL (k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

[34] Vaclav Pech, Alex Shatalin, and Markus Voelter. “JetBrains MPS as a tool for extending Java”. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM. 2013, pp. 165–168.

[35] *Peg.js*. URL: https://pegjs.org/ (visited on 04/18/2018).

[36] *Peg.js-fn*. URL: http://shamansir.github.io/pegjs-fn/ (visited on 04/18/2018).

[37] Eric Reed. “Patina: A formalization of the Rust programming language”. In: *Tech. Rep. UW-CSE-15–03-02* (2015).

[38] *Rust Playground*. URL: https://play.rust-lang.org/ (visited on 02/07/2018).

[39] *Scastie*. URL: https://scastie.scala-lang.org/ (visited on 03/26/2018).

[40] *Selenium - Web Browser Automation*. URL: https://www.seleniumhq.org/ (visited on 04/22/2018).

[41] *Self-hosting compiler for the Grace programming language*. URL: https://github.com/gracelang/minigrace/ (visited on 31/01/2018).

[42] Anthony M Sloane et al. “Monto: A disintegrated development environment”. In: *International Conference on Software Language Engineering*. Springer. 2014, pp. 211–220.

[43] Gianluca Stivan, Andrea Peruffo, and Philipp Haller. “Akka. js: towards a portable actor runtime environment”. In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM. 2015, pp. 57–64.

[44] Rabbelier Sverre. “Declarative Specification of Web-based Integrated Development Environments”. MA thesis. the Netherlands: Delft University of Technology, 2011.

[45] *The Go Playground*. URL: https://play.golang.org/ (visited on 02/07/2018).

[46] Masaru Tomita. *Generalized LR parsing*. Springer Science & Business Media, 2012.

[47] Vlad Vergu, Pierre Neron, and Eelco Visser. “DynSem: A DSL for dynamic semantics specification”. In: *Technical Report Series TUD-SERG-2015-003* (2015).

[48] Eelco Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[49] Eelco Visser et al. *Syntax definition for language prototyping*. Eelco Visser, 1997.

[50] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. "Building program optimizers with rewriting strategies". In: *ACM Sigplan Notices*. Vol. 34. 1. ACM. 1998, pp. 13–26.

[51] RG Vogelij. "Generating Web-based Semantically Aware Source Code Editors". MA thesis. the Netherlands: Delft University of Technology, 2013.

[52] *WebAssembly Goals*. URL: http://webassembly.org/docs/high-level-goals/ (visited on 09/28/2017).

[53] *WebDriverIO*. URL: http://webdriver.io/ (visited on 04/22/2018).

[54] Christian Wimmer and Thomas Würthinger. "Truffle: a self-optimizing runtime system". In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM. 2012, pp. 13–14.

[55] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, pp. 301–312.