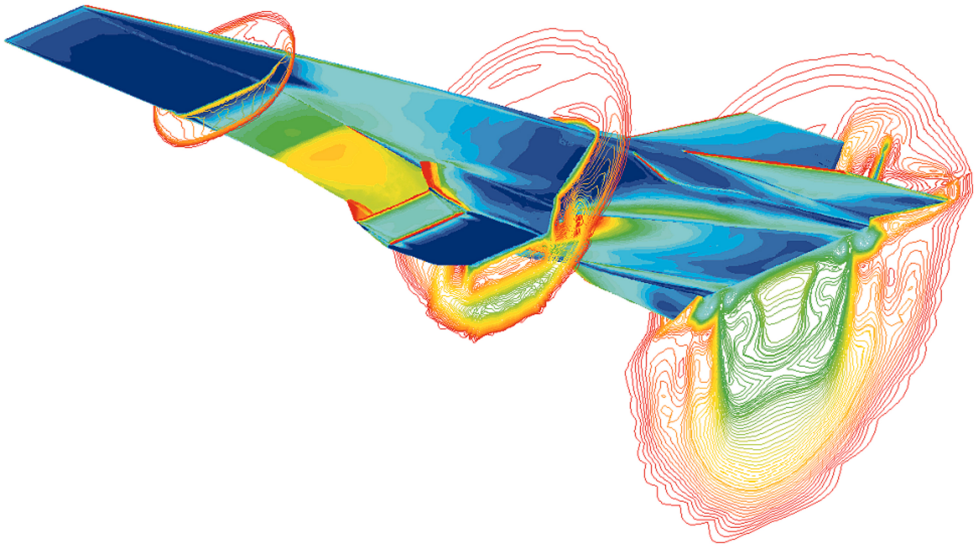
Dynamix on the Frame VM

Declarative dynamic semantics on a VM using scopes as frames



Chiel Bruin

Dynamix on the Frame VM

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Chiel Bruin
born in Alkmaar, the Netherlands

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

Cover picture: X-43A Mach 7 computational fluid dynamic (Wikimedia Commons).

Source code available on GitHub:

- The Frame VM: `https://github.com/MetaBorgCube/framevm`
- Dynamix: `https://github.com/metaborg/metaborg-dynamix`
- Tiger: `https://github.com/ChielBruin/metaborg-tiger/tree/exceptions`
- Scheme: `https://github.com/MetaBorgCube/metaborg-scheme`

Dynamix on the Frame VM

Author: Chiel Bruin
Student id: 4368436
Email: chiel@bruin.ch

Abstract

Over the years virtual machines (VMs) have been created to abstract over computer hardware. This simplified code generation and allowed for easy portability between hardware platforms. These VMs are however highly tailored to a particular runtime model. This improves the execution speed, but places restrictions on the types of languages that the VM supports.

In this thesis the Frame VM was developed as a VM that supports many different types of languages in a principled way. Achieving this is done by basing the VM on language independent models of memory and control flow. Usage of the scopes-as-frames paradigm and control frames resulted in an instruction set that is relatively small at its core, but does allow for the construction of complex control flow. As an effect, many different programming languages can be compiled to the Frame VM.

In addition to this VM, a Domain Specific Language (DSL) for executable semantics of programming languages was created. This language, Dynamix, allows for a modular approach to writing the semantics of a language. Additionally, Dynamix provides a meta-compiler that uses these semantics of a language to compile programs to the Frame VM.

To validate the Frame VM, direct compilers for Rust and Prolog have been created in a student project and compilers for Scheme and Tiger were created using Dynamix. Using these semantics of Scheme and Tiger, it was possible to execute programs containing usage of call/cc and a suite of Tiger benchmark programs.

Furthermore, the control flow of Tiger was extended with exceptions and generator functions. This extension did not require any changes to the existing semantics, showing the modularity of control achieved when using Dynamix and the Frame VM.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. S. Rellermeyer, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. P. D. Mosses, Dept. of Computer Science, Swansea University
University Supervisor:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft

Preface

I would like to thank all the people that made this thesis possible. Without their ideas, support and distractions, this thesis would certainly not have evolved into its current form.

Most importantly, I would like to thank my supervisors Casper Bach Poulsen and Eelco Visser for their guidance in directing the research and their help in structuring this report. I'm also greatly thankful for the feedback they and Andrew Tolmach provided to my early prototypes. Especially their "Isn't this equivalent to *X*?" and "Couldn't we just do *Y*?" questions helped in finding better abstractions and developing a deeper understanding.

I would also like to thank the Language Engineering Project (LEP) students Luka, Bram, Emiel and Fabian. I am grateful for both their courage in wanting to work with my implementation which was, at that point, very much in progress and for the feedback that they provided in doing so.

A special thanks goes to the (master) students who worked at the fourth floor, especially Taico, Maarten, the two Martijns, Caro, Bram, Luka, Dereck and Chris. They provided me with welcome distractions when I needed that, provided daily structure, allowed me to better focus on my work and gave me helpful feedback.

An other special thanks goes to those who provided feedback on the many drafts of this report. However rough the drafts were, their willingness to spend the time and their curiosity into what I had been working on greatly encouraged me.

Furthermore, I would like to express my great appreciation to those who pushed me to finish the writing of this report. Without the *vierde verdieping matties*, Kevin and my family, I would still be writing next year.

Lastly I would like to thank Eelco for allowing me to visit SPLASH in Athens. The conference provided me with useful insights, both personally and academically. Of those who also attended this conference, I would especially like to thank Jeff, Maarten, Ana, the other (old) members of the research group, Davina, Sam, Annabel and Lily for the great time.

Chiel Bruin
Delft, the Netherlands
March 27, 2020

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

2 Modeling Memory 5
2.1 Using Memory . . . 5
2.2 Scope Graphs . . . 6
2.3 Data Objects . . . 7
2.4 Intermediates . . . 9

3 Modeling Control 11
3.1 Compiling Complexity of Control Flow . . . 11
3.2 Continuations . . . 12
3.3 Control Frames . . . 15
3.4 Using Control Frames to Model Control Flow . . . 16

4 The Frame VM 19
4.1 Roger Syntax . . . 19
4.2 Initial Machine State . . . 21
4.3 Examples . . . 21
4.4 Semantics . . . 27

5 Language Engineering Project 41
5.1 Compiling Prolog . . . 41
5.2 Compiling Rust . . . 42
5.3 Frame VM Improvements . . . 43
5.4 Frame VM Applicability . . . 45

6 Dynamix 47
6.1 Syntax . . . 47
6.2 Compiling Tiger . . . 49
6.3 Library Functions . . . 56
6.4 Workflow . . . 57
6.5 Semantics . . . 58
6.6 Implementation Differences . . . 76

7 Scheme and Tiger in Dynamix **79**
7.1 Scheme . . . 79
7.2 Tiger Dynamix Specification . . . 83

8 Related work **91**
8.1 Compiling With Continuations . . . 91
8.2 Semantics Frameworks . . . 92
8.3 Meta-Programming . . . 92

9 Future Work **93**
9.1 Refine Control Frames . . . 93
9.2 Frame VM Improvements . . . 96
9.3 Dynamix . . . 97

10 Conclusion **99**

Bibliography **101**

Acronyms **105**

Glossary **107**

A Additional Frame VM Syntax and Semantics **109**

B Dynamix Primitive Frame VM Operations **113**

C Full Dynamix Specification of Scheme **117**

D Full Dynamix Specification of Tiger **123**

E Dynamix Library Rules **133**

List of Figures

2.1 Similarities between let-bindings, functions and variable declarations. 5
2.2 Scope graph and reference resolution. 6
2.3 Applying scopes-as-frames to a scope graph. 7
2.4 Frame representations of common data structures. 8

3.1 Different types of control flow 12
3.2 CPS transforming integer addition 13
3.3 CPS safe integer division 13
3.4 Call/cc in Scheme 14
3.5 More complex call/cc example in Scheme 15
3.6 Control graphs of control flow constructs. 17
3.7 Control graphs of control flow constructs (cont.). 17

4.1 A code block written in Roger. 19
4.2 Roger syntax 20
4.3 A *Hello World!* program on the Frame VM 21
4.4 Constructing data objects in Roger. 22
4.5 An example of an if-statement in Roger 23
4.6 A while-loop in a Roger program 24
4.7 Function calls on the Frame VM. 25
4.8 A program that throws and catches an exception written in Roger 26
4.9 The Roger core syntax 28
4.10 Desugaring rules for complex Roger operations. 29
4.11 Transforming a Roger program to the core language. 30
4.12 Additional notation used in the semantic rules of the Frame VM 31
4.13 Projection functions used in the semantic specification of Roger 31
4.14 Semantic rules for Frame VM initialization 32
4.15 Semantic rules for the Frame VM main evaluation loop and termination 33
4.16 Semantic rules for evaluation of code blocks 33
4.17 Semantic rules for evaluating instructions 34
4.18 Semantic rules for evaluating control-influencing instructions 35
4.19 Semantic rules for basic operations 36
4.20 Semantic rules for comparing integers and checking the type of a primitive value 37
4.21 Semantic rules for complex value creation. 37
4.22 Semantic rules for expressions that operate on data frames. 38
4.23 Semantic rules for miscellaneous expressions. 38

5.1 Pseudocode of execution of a Prolog rule 42

5.2 Rust object representation . . . 42
5.3 The syntax of using imported functions in Roger. . . . 43

6.1 Syntax specification of Dynamix . . . 48
6.2 A mapping from Dynamix operations to Frame VM instructions . . . 49
6.3 Built-in Dynamix primitive operations . . . 49
6.4 The `eval` rule of the Tiger compiler . . . 50
6.5 The `eval-exp` rules of some simple Tiger expressions . . . 50
6.6 The `eval-exp` rules of let bindings in Tiger . . . 52
6.7 The `eval-exp` rules of an if expression . . . 53
6.8 Evaluation rules for a while-loop in Tiger . . . 54
6.9 The semantics of functions in Tiger . . . 55
6.10 Dynamix helper functions for function calls . . . 56
6.11 Dynamix development workflow . . . 58
6.12 Transformation rules to core Dynamix language. . . . 59
6.13 Separating the two different Dynamix bindings. . . . 60
6.14 Additional notation used in the semantic rules of Dynamix . . . 61
6.15 Projection and update functions used in the semantics of Dynamix. . . . 61
6.16 Initial meta-compiler invocation. . . . 62
6.17 Rule matching . . . 63
6.18 Rule matcher helper rules . . . 63
6.19 The first set of matching rules for the rule arguments. . . . 64
6.20 The second set of matching rules for the rule arguments. . . . 65
6.21 Semantic rules for compiling the body of a Dynamix rule. . . . 66
6.22 Semantic rules for quoted block pre-allocation. . . . 66
6.23 Compiling the compile-time bindings of the Dynamix rule. . . . 67
6.24 Semantic rules for the runtime bindings and expression statements. . . . 68
6.25 Last step of rule body execution. . . . 69
6.26 Semantic rules for evaluation of variable references and term construction . . . 70
6.27 Evaluation rules for call expressions. . . . 71
6.28 Semantic rules for rule argument evaluation. . . . 72
6.29 Semantic rules for constructing terms. . . . 73
6.30 Miscellaneous Dynamix primitive operations and their semantics. . . . 74
6.31 Semantics of Dynamix primitive scope graph operations . . . 75

7.1 Detailed call/cc behavior . . . 80
7.2 Closure creation using Dynamix . . . 81
7.3 Dynamix implementation of `call/cc` . . . 82
7.4 Dynamix implementation of Scheme REPL simulation. . . . 82
7.5 Tiger benchmark results . . . 84
7.6 The extension of the Tiger syntax with exceptions . . . 85
7.7 Dynamix rules for Tiger exceptions . . . 86
7.8 Dynamix rules for safe division . . . 86
7.9 The extension of the Tiger syntax with generator functions . . . 87
7.10 Dynamix rules for generator functions . . . 88
7.11 Dynamix rules for for-in loops . . . 89

9.1 Delimited continuations example . . . 94
9.2 Modeling delimited continuations using control frames. . . . 94
9.3 Construction of a scoped control graph. . . . 95

A.1 Additional Roger syntax. . . . 109

A.2 Desugaring additional Frame VM syntax. 110
A.3 Desugaring additional Frame VM syntax (cont.). 111
A.4 Additional semantic rules for Roger. 112

B.1 Semantics of Dynamix primitives for data frame operations 113
B.2 Semantics of Dynamix primitives for control operations 113
B.3 Semantics of Dynamix primitives for primitive value creation and debugging . . 114
B.4 Semantics of Dynamix primitives for integer operations and data comparison . . 114
B.5 Semantics of Dynamix primitives for control frame operations 115

C.1 Full Dynamix specification of Scheme. 121

D.1 Full Dynamix specification of Tiger. 132

E.1 Dynamix specification of library functions used in Appendix C and D. 137

Chapter 1

Introduction

When comparing the first electronic computers to today's smartphones, gaming consoles and supercomputers, it is clear computers changed a lot since their inception. Not only has hardware become faster, cheaper and more efficient, today's computers are ubiquitous and used for ever more complex tasks.

A trend that is not directly visible is in the software programs computers run, more specifically, the programming languages these programs are written in. Over time, these languages introduced increasingly higher levels of abstractions [52]. From machine languages that used processor opcodes and memory locations for programming the earliest machines, programming languages evolved into languages like Java and Python. These new languages have resulted in productivity improvements of several orders of magnitude [44].

One of the problems of machine languages is its usage of absolute memory locations to refer to other sections of a program. Modifying a program by adding or removing instructions at other locations then the end, would therefore invalidate large sections of a program. The first assemblers and assembly languages were invented to take care of exactly this problem [40]. Using labels, abstractions were added to the machine language to abstract over their exact locations in the code. Besides the addresses, these assembly languages also abstracted over the opcodes by using more readable names for the operations [52].

In the early 1950s the first high-level programming languages were designed[1] [52]. These languages included ever more levels of abstraction over the machine language. Memory addresses were replaced by variables and assignments, control structures were added to replace jumps, subroutines made way for procedures and functions, and encapsulation mechanisms like packages were introduced to improve program modularity [52].

These evolving languages also resulted in various paradigm shifts, which in turn produced new programming languages. Procedure-oriented programs were replaced by data-oriented designs in the late 1970s, which soon evolved in to the object oriented programming paradigm [40].

Besides adding more abstractions to the languages themselves, computer scientists also increased the level of abstraction of the compilation targets. This simplifies the task of writing a compiler by, for example, providing support for different machine languages (e.g. x86, ARM) and optimizations. For example, many programming languages use C as an intermediate language and use GCC for the compilation to machine code [26]. As C was not originally designed for this purpose and therefore has its limitations, other languages have been created to act as such an intermediate assembly language [26].

Going one step further, virtual machines (VMs) completely replace machine language. A VM defines its own byte code language containing more high-level programming concepts.

[1] When compared to assembly languages. Compared to modern high-level languages these are still fairly low-level.

This language is then interpreted by the VM ensuring portability to any machine that has a byte code interpreter and runtime [40].

Like how computer architecture had a great influence on programming language design [40], intermediate languages and VMs had a big influence on the languages that target them. Such systems embody many design decisions, making that the semantics of a language must fit those of the intermediate language or VM. But as different languages use different object layouts or control flow mechanisms, this semantic match is not guaranteed [26].

Naively this problem could be solved by adding features for all different types of operations that languages perform. However, like what happened to the language PL/I, this would result in a huge, complex and incoherent design. A systematic approach of combining smaller language concepts would result in a better, more coherent system [52].

In this thesis an exploration is made into finding the fundamental components of executing programming languages. These components could then be combined to form a principled and relatively small instruction set for a VM (called the Frame VM).

Creating such a general solution does however come with a trade-off. Supporting many language features removes the possibility to tinker with the design for a specific use-case. This has the effect that, while it can perform more general operations, the execution speed can never reach the level of a targeted runtime.

The trade-off made between speed and general applicability is however justified by the target application of the VM. Instead of this application requiring the absolute fastest execution speeds, a solution that is built from principled building blocks that allow well structured execution of various different programming languages is preferred. This is because the VM is designed to be used in tight cooperation with the Spoofax language workbench [27].

In this workbench, new programming languages can be created, or existing languages can be described, using Domain Specific Languages (DSLs) describing various stages in the compiler workflow. Using these descriptions, for example the syntax of the language and its name binding rules, a complete Integrated Development Environment (IDE) is generated for the described language. In such a language workbench, execution speed is not the initial concern. Rapid language development and support for all kinds of language constructs is far more important. Later stages of the language development might include the creation of a dedicated, high-performance compiler, but in the initial stages it might not even be yet clear what the syntax of the language will become, let alone that the best way to compile it can be envisioned.

The VM that is created during this thesis, and is described in this report, is therefore designed around formalizations of components of program execution. By using these formalizations of, most importantly, control flow and data/memory allocation, the VM is able to execute many different programming languages, while maintaining a structured approach to the execution.

Targeted at these two components, two main research questions guided the research in this thesis. The first was *whether the scopes-as-frames-paradigm could be used to create (the instruction set of) a VM*, giving it a language independent memory model. The second question was *which general model for control flow could be used by this VM*, where this model should support very general control flow constructs like `call/cc`.

Furthermore there was the question of *how meta-compilation could be used to compile a program to the VM*. This meta-compilation would use the semantics of the compiled language to compile a source file to machine instructions. The language Dynamix, introduced in this thesis, uses this technique to bridge the gap between the declarative language descriptions in the Spoofax language workbench and the compiled programs running on the VM.

The contributions made in this thesis include:

- An overview of formalizations for computer memory and data structures (chapter 2)
- A formalization of runtime control flow using control frames (chapter 3)
- The Frame VM, that uses these formalizations in the core of its design (chapter 4)
  - A formal syntax definition of its instruction set (section 4.1)
  - A formal definition of its semantics (section 4.4)
- Dynamix, a DSL for executable semantics descriptions of languages (chapter 6)
  - A formal syntax definition of Dynamix (section 6.1)
  - A formal definition of its semantics (section 6.5)
  - A description of the semantics of Tiger and Scheme (section 6.2 and chapter 7)
- A demonstration of compiling call/cc to the Frame VM (section 7.1)
- A demonstration of modular control flow semantics (section 7.2)

Chapter 2

Modeling Memory

In the execution of a computer program, memory is ubiquitous. Abstracted away by language constructs and compilers, data is continuously being moved between CPU registers, system memory (RAM), disk and caches. This chapter describes a number of formalizations of these abstractions that allow for a language-independent description of the runtime memory used by a program.

To start, section 2.1 makes a distinction between different types of abstractions in programming languages and the usage of memory. Later sections give formalization methods that can be used to describe these categories in a language-independent way.

2.1 Using Memory

Programming languages contain various constructs that describe memory (e.g. let-bindings, variable declarations, objects, arrays, records, function/procedure arguments). Some of these are clearly distinct concepts, while others have a lot in common. For example, a variable declaration is clearly not the same as an array. But let-bindings and variable declarations are both describing different variations of binding (Figure 2.1).

It is possible to group all these language constructs into two distinct groups. The first group are the abstractions over *where* data is stored and the second contains the constructs on *what* data is stored:

- **Variable bindings** (*e.g. let-bindings, variable declarations, function/procedure arguments*)
- **Data objects** (*e.g. objects, arrays, records*)

These two categories can be used to describe most abstractions for memory used in programming languages. But there is one more distinction that has to be made in order to describe all aspects of how a program uses memory. Consider for example the calculations $x := 1 + 2$ and $x + (3 + 4)$. Here the value for x and the result of $(3 + 4)$ are both integer

```
(let ([a 1]
      [b 2])
  (let ([c 12])
    (+ a (* b c))
  )
)
```

```
def func (a=1, b=2):

    c = 12
    return a + (b * c)
```

Figure 2.1: Similarities between let-bindings, functions and variable declarations.

```
def func(a, b):
  return a + b

def bar()
  return func(1, 2)

bar()
```

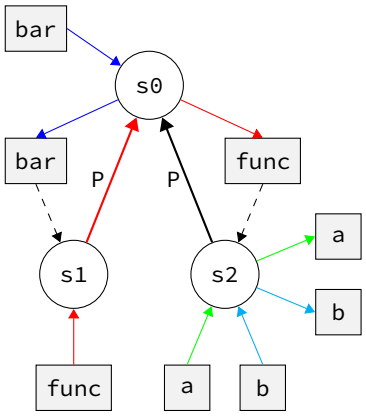


Figure 2.2: The scoping and name resolution of a program expressed by its scope graph. The colored paths show how each reference resolves to its declaration.

values and are both present at runtime. They are however treated in a completely different way.

This is because x is part of the family of (named) local variables and the result of $(3 + 4)$ is not. Local variables are values that can be reused multiple times and can, optionally, take different values.

The result of $(3 + 4)$ is clearly not a named local variable. On the contrary, it is an anonymous intermediate value. These are values that are not named in the source code and are used only once [15].

In broad strokes, though not in general, local variables are stored on the stack or on the heap in most computers. Furthermore, these values are no longer of interest when they go out of scope [15]. The anonymous variables are mostly kept in the processors registers for quick access, with additional values spilling to the stack.

2.2 Scope Graphs

A formalization that nicely describes the category of variable bindings are scope graphs. Scope graphs, as introduced by Neron et al. [35], are a language independent model to describe relations between references, declarations, scopes and imports [35]. In Figure 2.2 an example is given of a program and its scope graph. On the right side of the figure, the corresponding scope graph of the program is shown including colored resolution paths.

In this graph all variable declaration and references from the program can be seen. In the root scope $(S0)$ the declaration of function names `func` and `bar` can be seen. Each of these declarations has an associated scope for their function bodies. In the case of `bar` its scope $(S1)$ only contains a single reference. This reference refers to the function `func`, of which the declaration can be found by traversing the directed edges of the graph. Scope $S2$ includes declarations for each of the function arguments. As an effect references to the argument values resolve to the correct declarations, even when a variable named `a` was already declared in a different scope.

There exists a clear connection between static name resolution and scoping on one side, and dynamic memory access and memory allocation/deallocation on the other side [7]. This connection is formalized in the scopes-as-frames paradigm. In this paradigm, the static layout of the scope graph is made to correspond to the dynamic (run-time) layout of frames on the heap [7]. Concretely, this means that for each instance of a scope in the scope graph there exists a heap-allocated frame with two basic properties: 1) The heap frame is linked to zero or more heap frames, in such a way that each linked frame corresponds to a parent

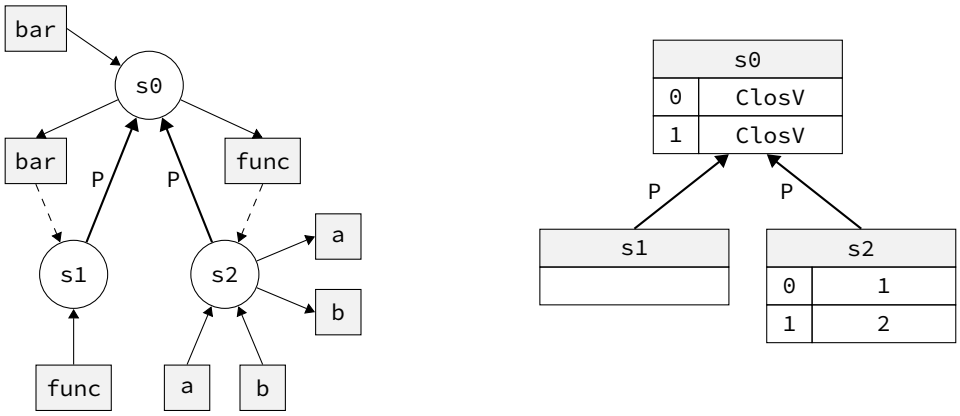


Figure 2.3: The scope graph from Figure 2.2 and the memory structure using the scopes-as-frames paradigm.

scope of the scope represented by the linking heap frame. 2) Each declaration of a name in the original scope corresponds to a memory region inside the heap frame (called a slot).

If you take the same program and scope graph from Figure 2.2 as an example and apply the scopes-as-frames paradigm to it, you get a memory layout as shown in 2.3. In this figure, the general layout of references, declarations and the resolution paths between them is still the same as in the original scope graph. As you can see in 2.3, getting the actual value of `a` is done by looking up the value stored in she slot corresponding to the name `a` in the current heap frame. This is similar to the static resolution path, where `a` resolves to its declaration in the current scope. Similarly, `func` statically resolves to its declaration in the parent scope of the current scope and getting its runtime value is a memory lookup of the link P of the current frame to get the parent frame, followed by loading the value from the correct slot of the that frame. In this case the value of `func` will be found to be a closure instance. The actual implementation of such a closure value is discussed in in chapter 7, as it is not important for now.

One key difference between frames and scopes from a scope graph is that scopes are static and that frames are dynamic instances of these scopes. A function call with a single static scope in the scope graph can therefore be represented by different frames during execution.

As the next chapter introduces control frames as a model for describing control flow, the word frame can be used to describe two different concepts. To remove this confusion, (heap) frames will be referred to as data frames from now on.

2.3 Data Objects

The second category of language constructs from the start of this chapter are data abstractions. Examples of this are records/objects, arrays and more complex data structures like trees. These data objects can also be modeled using a scope graph. For example, a record type `Point` with an x-coordinate `x` would be modeled as a scope containing a declaration of `x`. This concept is called *scopes-as-types* [3].

It is simple to think of an implementation of a `Point`-record using the data frames from the last section. As a `Point` is a record containing two coordinates, we could take a frame with two slots to represent an instance of this record (Figure 2.4). Again, we see that the memory layout created using this method corresponds nicely with the scope graph we get when modeling the record type as a scope. In most cases, scopes-as-types and scopes-as-frames can transitively be applied in this way to directly transform a data object to a data frame/ collection of data frames.

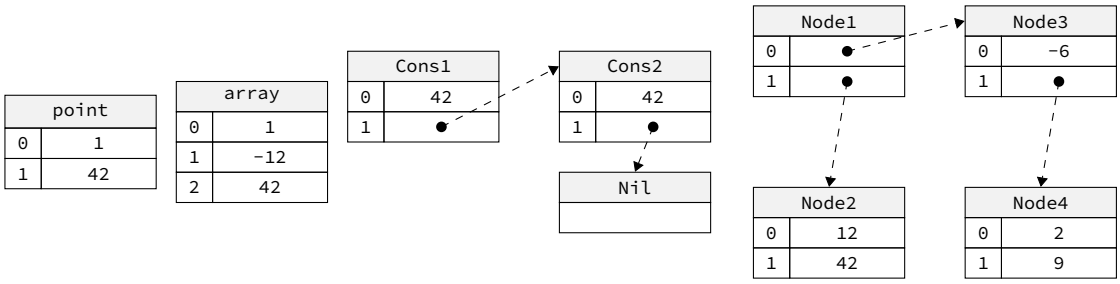


Figure 2.4: Frame representations of common data structures. From left to right: The point record, arrays, cons-lists and a search tree.

In some cases, this correspondence between scopes and frames does not entirely hold. For example, an array cannot be expressed in terms of a static scope graph. The reason for this is that the length of the array cannot always be statically known. But, even if we did know the size statically, the indices inside the array are not easily encoded in a scope graph. Looking at the scopes-as-frames paradigm and the way we would create an array using a frame with a slot for each index, we would model the array as a scope containing a declaration for each index. These declarations are not indexed and require a name. The indices could be taken as a name, but this is arguably a bit of a hack.

Other dynamic data structures like cons-lists and trees can also easily be described by using data frames(see Figure 2.4). In the case of a cons-list each element of the list is stored in a data frame containing the head value and a reference to a list tail. For a tree a similar construction is used where each node is a data frame which contains references to its children, or a value at a leaf. But again, no clear correspondence to a static scope graph can be given for them.

2.3.1 Good Heap Property

In the paper by Bach Poulsen et al. [7], the good heap and good frame properties are described. When these properties hold for a heap, they state that each data frame on the heap is well bound and well typed. A data frame being well bound means that the data frame maintains the same structure as the scope graph. Being well typed implies that the value stored in a data frame slot has the same type as the declaration in the scope graph. Combined, these properties can be used in constructing type soundness proofs.

With some dynamic data types not completely fitting the scopes-as-frames paradigm, not all data frames have these properties. For example, a data frame representing an array has no scope in the scope graph, making that the well bound property does not apply.

One could propose a refinement of the well bound and well typed properties to take care of these cases where they cannot be applied. Such a refinement might be to split the heap in two sub-heaps, one for the data frames that adhere to these properties and one for those that don't. However, there is likely a more fundamental solution to this problem.

At the core of the problem lies the fact that there are data frames that cannot be expressed in the scope graph. Not being able to apply the properties to those data frames is just an effect of this problem. A more fundamental solution would therefore look into ways of modeling these dynamic data types as a scope.

For modeling homogeneous arrays, it could suffice to add something like an array scope. Such a scope would allow for a dynamic size, and could have an attached type that is correct for all elements in the array. With this, the well typed property could directly be applied and only a small change would be required to the well bound property to handle dynamic sizes.

For other data types, like cons-lists, this solution does not work. These data types have the problem that, over time, contained elements could have different types. For example, the element stored in the tail slot of a link in a cons-list could either be the tail or an other link. To apply the well bound and well typed properties, this either relation should be expressible in the type of a declaration. This would mean that the scopes-as-types paradigm needs to be extended with an either relation.

More research is however needed towards such a refinement of the scope graph model. Adding array scopes and allowing an either relation in scopes-as-types seem to be able to encode these more dynamic data types while retaining support for the good heap property. However, the full implications of these additions and whether they are a complete solution has not yet been investigated.

2.4 Intermediates

In contrast to variable bindings, anonymous intermediates do not depend on the structure of a program, but rather are a product of the semantics of a language. The program itself does not specify where values should go in an expression like $(1 + 2)$, but the semantics specify that the 1 is evaluated first and its result will be stored some temporary variable (like a CPU register).

A second way in which intermediates contrast variable bindings is that anonymous intermediate values do not have a nice formalism, like scope graphs, that describes their structure and behavior. Rather the handling of intermediate values is done in either of two ways. The first stays close to the implementation in hardware by using a register machine for handling intermediate values. These registers either directly map to registers in a processor (and potentially spill to the stack), or an intermediate naming scheme is used. The names used in the latter case are then mapped to actual registers by the runtime environment/compiler.

A second approach is to have a (virtual) stack for intermediate values. Values can be pushed onto this stack to store them and they can be popped of the stack to use them. Using this stack for storage of intermediate values removes the need for complicated register allocation algorithms, with the expense of more data copy instructions [13].

Chapter 3

Modeling Control

Scope graphs have been proven a useful tool in modeling the binding of names in a program. Using the scopes-as-frames-paradigm this model can be converted to a language language-independent description of the runtime memory of a program. By providing this runtime memory model, this paradigm describes what is essentially an API or instruction set for the memory of a general machine.

One of the questions that initiated the work described in this thesis, was whether such a model, and such an instruction set, could be created for the control flow of a program. This chapter introduces the control frame as the basis of such a model. These frames act as building blocks of control (flow) and using these building blocks, and the operations that can be performed on them, the behavior can be expressed of arbitrary programming language constructs.

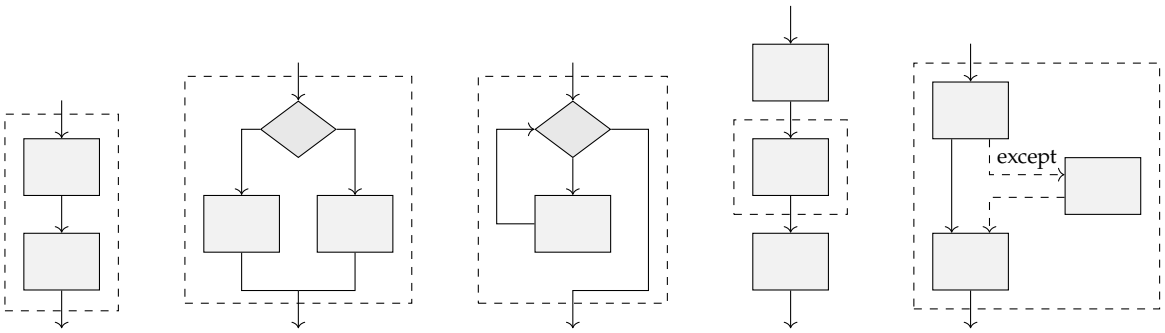
To establish some groundwork on the behavior of control flow constructs, section 3.1 lists a number of common constructs and how they are compiled to machine instructions. Using these compilation steps and some theory on continuations (section 3.2), section 3.3 introduces control frames as a runtime model for the control flow of a program. Similarly to how scopes in a scope graph are used to mean a minimal region in a program that behaves uniformly with respect to name resolution [35], a control frame describes a program region that behaves uniformly with respect to control. As is shown in various examples in section 3.4, this construction provides an elegant solution for describing control flow constructs like function calls and exception handling.

3.1 Compiling Complexity of Control Flow

When the goal is to derive a language-independent machine model for control flow, it is beneficial to first look at various types of control flow constructions that are used in programs. Therefore this section will look at increasingly more complex constructs and how those are conventionally compiled and executed. The flow of these constructs is shown in Figure 3.1 as a partial Control Flow Graph (CFG) where the box shows surrounds the nodes related to the construct.

The simplest form of control flow in a program is the linear/sequential flow. In a conventional machine linear control is performed using a linear list of machine instructions and the program counter (PC). This counter increments after execution of each instruction, resulting in the machine instructions being executed consecutively.

Conditional statements, and more generally jumps, allow this PC to be modified at runtime. As an effect the machine instructions can be skipped or executed multiple times. In the case of conditionals like if-statements this means that the PC will be updated to point to the instructions of the then-branch when the condition evaluates to true or to the else-branch when false.



(a) Linear (b) Conditionals (c) Loops (d) Functions (e) Exceptions

Figure 3.1: Different types of control flow

Compilation of a looping construct like a while loop also uses jump instructions that update the value of the PC. When the loop condition is true, the PC is updated to point to the instructions for the body of the loop. At the end of the loop body, this pointer is reset to the location of the loop termination check. When the condition still holds true, the body is executed one more time. But when the condition fails the PC is updated to point to the code that is executed after loop termination.

However, this technique of updating the PC to static code locations is not how more complex control flow constructs can be compiled. Take for example a function call. At some point after calling a function, the function needs to return to its call site. Naively the PC can again be updated to point to correct the instructions. But for this to work some extra data is required, as the return address is not statically known. In a recursive function the return address can, for example, either point back into the same function or to the original caller.

This problem of the return addresses being dynamic is solved by the call stack. On this stack call frames are stored that store a dynamic link to the caller of a function [16]. When a function is called, a new frame is pushed onto this stack and on a return this frame is popped from the stack [52]. This stack of dynamic links then allows for resolving the correct return addresses of a function.

When exceptions are added to a language, the compilation again becomes more complex. On raising an exception, the code must jump execution to the closest exception handler. Keeping track of where this handler is also requires the call stack. When an exception is raised the stack then needs to be unwound to find the exception handler. This unwinding keeps popping stack frames from the stack until a stack frame for an exception handler is found. The call frames popped in this procedure can then be used to construct a trace to the origin of the exception.

Some languages also allow execution to be resumed after an exception is handled. But if execution can be resumed after an exception is raised, the stack unwinding must be deferred [29]. This further complicates the function of the call stack.

For even more complex control flow constructs, most notably those that make use of continuations, a simple stack does not even suffice. Most Scheme implementations, for example, need to use the heap for storing their call frames. Dybvig was able to optimize this model back to using a call stack, with the consequence that snapshots of the stack had to be created when continuations were created [16].

3.2 Continuations

When describing the control flow of a program, continuations can be a very useful tool. The continuation of some calculation specifies a state transformation for the remainder of the

```
(+ (+ 1  2) 3)

(+' 1 2 (lambda (x) (+' x 3 id)))
```

```
(define +'
  (lambda (e1, e2, k)
    (k (+ e1 e2))
  )
)
```

Figure 3.2: An example of transforming an expression adding three integers.

computation from a given point. Simply put, a continuation expresses "what to do next" in its execution [4].

When calculating $1 + 2 + 3$, the continuation of the sub-expression $1 + 2$ is to add 3 to its result. A continuation can therefore be seen as an expression with a hole that needs to be plugged with some other expression or value [12]. The continuation of $1+2$ could therefore be seen as $\bullet + 3$. The intermediate result 3 then plugs the hole creating the next calculation step $3 + 3$. Similarly, the continuation of $(2 + 3)$ in $1 + (2 + 3)$ is $1 + \bullet$. The location of the hole now changed with respect to the evaluation order of the complete expression, exposing some important information about the control flow of the program.

Using continuation passing style (CPS) all aspects of control flow and data flow can be made explicit [4]. In this style of notation the continuation of an expression is no longer described using a hole, but using an function that given the intermediate result will compute the result of the complete computation. The expression $1+ \bullet$ from before thus becomes $\lambda v.1+ v$. This function is then made explicit in the complete calculation, transforming $1 + (2 + 3)$ into $(\lambda v.v + 1)(2 + 3)$.

In practice, a slightly different notation is used. This notation puts the continuation expression as the last argument to a function application. An example of this notation is shown in Figure 3.2. In this example, the $+$ function is defined to first evaluate the addition of the two first arguments to some intermediate value followed by a call to the continuation.

While this is not the most exciting example on CPS that could be shown, it does convey the core concepts of passing the remainder of the computation around in order to make control and data flow explicit. In addition, this simple example already shows that there is some relation between the return address of the addition function and the continuation passed to the addition function in CPS. We can make this relation even more clear if we create a CPS transformed function which has multiple return addresses. Figure 3.3 shows a CPS transformed division function with an exception handler that gets invoked on a division by zero. Now we have to provide a call to this division operator with not just one, but two continuations. One for the case that the division succeeds, and one for the case in which it results in an error.

```
(define /'
  (lambda (e1, e2, k, x)
    (if (= e2 0)
      (x e1)
      (k (/ e1 e2)))
    )
)

(/' 42 0 (lambda (v) (print v)) (lambda (x) (print "division by zero")))
```

Figure 3.3: An example of a function that performs safe integer division using CPS

```
(define add-2 #f)                    ; -
(+ 2                                 ;
  (call-with-current-continuation    ;
    (lambda (cont)                   ;
      (set! add-2 cont)              ;
      3                              ;
    )                                ;
  )                                  ;
)                                    ; 5
(add-2 40)                           ; 42
```

Figure 3.4: An example of call-with-current-continuation in Scheme. The results are shown as comments.

From this example it is clear that a CPS transformation makes the control flow of a program very explicit. This information being explicit has great benefits in many applications, like compiler optimizations. Therefore an intermediate language based on CPS is used in a number of compilers. It is believed that it provides for better data flow analysis, eases optimizations and simplifies code generation [38]. In fact, a CPS-like transformation was first described by Adriaan van Wijngaarden as a means of simplifying code generation. This was years before the concept of continuations was coined by C. P. Wadsworth [36].

3.2.1 Call/CC

One of the most interesting applications of continuations, is when the current continuation is exposed to the programmer. Probably the most well known language that allows this is Scheme with its `call-with-current-continuation` procedure (or `call/cc` for short). This procedure wraps the current continuation in a procedure and provides this as an argument to the enclosed lambda.

A "simple" example of the usage of `call-with-current-continuation` is shown in Figure 3.4. The continuation of this program at the location where the continuation is captured is to finish the addition of 2 to the result of the call/cc call. If we store this continuation (as is done on line 5), we basically stored a function that adds 2 to whatever it is given as an argument. Indeed, when the continuation is invoked on the last line, the result is an addition of the provided argument and the number 2.

Having access to a `call/cc`-like procedure allows programmers to write their programs in continuation passing style (CPS). As an effect, `call/cc` can be used to model various language constructs like early function returns, loop breaks, co-routines and backtracking [28].

One important detail that was glossed over, with respect to the definition of continuations, was the evaluation context. In most CPS-transformed programs the way in which this context is handled is not very important. But when a language contains a construct like `call/cc`, it cannot be ignored. Consider the example from Figure 3.4, but instead of 2 a value x is added to the result (Figure 3.5). In this case the result of evaluating the continuation depends on the value of x that is stored in a context. When seeing the continuation as an expression with a hole ($\bullet + x$), the information about x is lost when this is stored as the continuation. Invoking this continuation when x is no longer bound would therefore result in an error. A correct definition would therefore be that a continuation describes not only the evaluation of a program from its current state to its final (terminal) state, but to also includes the semantic context (i.e., the environment and store)[46].

```
(define add-x #f)                    ; -
(let (x 2)                           ; -
  (+                                 ;
    (call-with-current-continuation  ;
      (lambda (cont)                 ;
        (set! add-x cont)            ;
        3                            ;
      )                              ;
    )                                ;
    x                                ;
  )                                  ; 5
)                                    ;
(add-x 40)                           ; 42
```

Figure 3.5: The example from Figure 3.4, but instead of 2 the value of x is added.

3.3 Control Frames

Fully transforming a program to continuation passing style (CPS) comes with the benefit that all aspects of control are made explicit. This would make CPS a good candidate for the theoretical grounds of a model that describes the control aspects of a program. There are however a few drawbacks in declaring CPS a universal building block of control. Most importantly, not all programming languages allow rewriting a program in CPS. This requires at least first class functions to model the remainder of a computation; a feature that is not present in all languages.

A smaller drawback of fully transforming a program to CPS is that it is really fine-grained. For example, it is not really necessary to use CPS to model the left to right (or right to left) evaluation order of simple expressions like integer addition. In such a case, CPS only increases the bookkeeping that is required in passing around the continuations. Any insights in the control flow of a program that could have been derived from the CPS are merely obfuscated by these fine-grained details.

In this thesis the *control frame* is proposed as a solution to this problem. A control frame describes the context of the evaluation of a minimal program region that behaves uniformly with respect to control flow. This context includes both the memory used during evaluation and the flow of control between different program regions.

A minimal program region, as described in this definition, is more coarse-grained than the program units in a fully CPS-transformed program. Examples of such a region are function calls or the branches of if statements. While the body of a function might perform a lot of different operations, the return address of the function is shared between all of the subexpressions. This makes the body of a function behave uniformly with respect to its return address and thus its control flow.

Definition 3.3.1. Control frame $cf = \langle cs, pc, s, \sigma \rangle$, where $cs = \{x \mapsto cf\}$, x is a continuation label, pc is the value stored in the program counter, s is a reference to a scope and σ is a set of registers storing intermediate values.

The context of evaluation that a control frame describes consists of a description of memory that is required for the upcoming calculations. This memory consists of two parts: Intermediate values, which are defined by the semantics of the language, and values that are defined in the program. The second class of these values is perfectly described by the scope graph of the program and the scopes-as-frames paradigm. Therefore the third component of a control frame is a reference to a scope in a scope graph. Intermediate values do not have

such a clear formalism. For now they are represented by a set of registers. These registers map variables used in the semantic description of a language to their values, and are the final component of a control frame.

Besides the context of evaluation, a control frame also stores the execution state and future control flow. An important part of this control flow is the current computation. In hardware, this current computations is represented by a program counter (PC) that stores the memory location of the instruction to execute. A control frame uses this same concept and therefore stores a pointer to some instruction that is currently executing.

For the future control flow, a control frame also contains a set of continuations. These continuations are used to keep track of various return addresses. In the case of a function, this set would thus contain a continuation of the return address of the function. Like in a CPS-transformed program, multiple continuations can be used to encode more complex control flow constructs. Using different continuation labels these can then be distinguished from each other.

From the definition of the set of continuations, it can be seen that a continuation is just an other control frame. When looking at the definition of a continuation at the end of section 3.2, we know that in order to encode a continuation two components are needed. The first part is the "what to do next"-part of the continuation. This can be encoded using the current PC and a set of return addresses of the current program region. The second component is the current semantic context. A description of the state of the currently used memory suffices for this. These four components required to describe a continuation match the contents of a control frame perfectly. Therefore a control frame can be used to encode a continuation.

3.4 Using Control Frames to Model Control Flow

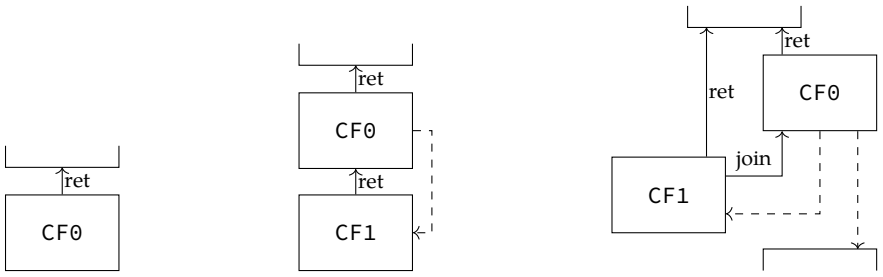
In the first section of this chapter, various control flow constructs were shown. Each of these constructs can also be expressed in terms of control frames. For this, a control graph is used as a visual representation of the structure of control frames. This graph is created by linking all control frames used in a program together using the continuations they store. In the resulting graph each node represents a control frame as a rectangular box. The continuations are shown as directed edges which, just like the continuations in a control frame are labeled.

The PC stored in a control frame is not shown in this graph. Optionally, a dashed arrow is sometimes drawn to a control frame that is created and called by another control frame as a means of describing the execution steps taken from the current PC. For example, when calling a function, this arrow points to the control frame for the body of the function.

Each node in the graph also has a directed edge from inside the rectangle to a scope in a scope graph. This edge is sometimes left out for simplicity, as including the entire scope graph makes the complete graph more complex. When the goal of the image is to describe just control flow, this extra information will therefore only make it harder to convey the ideas.

The values stored in a register do not add to the structure of the control graph. For this reason, the register values are omitted from the graph most of the time. When included, the control frame rectangle is expanded to include an indexed list of values.

The first type of control flow that was described in section 3.1 was the linear flow of control. Each instruction in this flow behaves uniformly with respect to control; they all evaluate in the same order and all share the same return address(es). Therefore a single control frame can be utilized to model linear control flow. Figure 3.6(a) shows this single control frame with its single return address. If the linear section was the body of a function, this would refer back to the caller of the function. When the linear section was at the top level of the program, the return continuation would signal a program exit.



(a) Linear (b) Function call (c) Looping

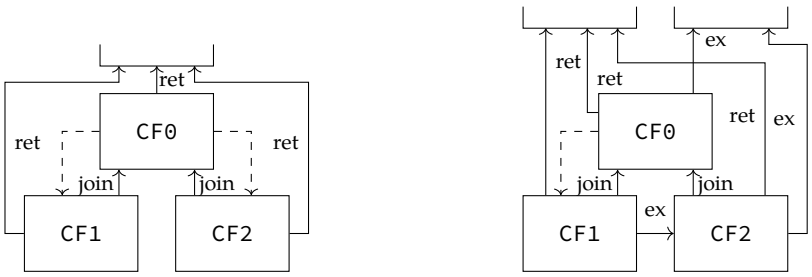
Figure 3.6: The control graph of the execution of a linear program, a function call and a looping construct.

From this it is a simple step to model a function call (Figure 3.6(b)). As the function body and the caller do not share the same return address, two control frames are required this time. The continuation of the control frame for the function body (`CF1`) is to return to the call site (`CF0`).

When modeling a loop it is no longer sufficient to use a single return address. First, the continuation of the body of a loop is to return to the condition check (`join`). Next, the body needs to retain the correct (function) return address (`ret`). This return address is the same as that of its parent control frame that performs the condition check. This way returning inside the loop body has the same effect as returning outside of the loop body. Optionally a third continuation can be added to the loop body to model a break.

The control graph for an if-statement (Figure 3.7(a)) is similar to that of a loop. The only difference is that there are three instead of two control frames now: One for each of the branches and one parent control frame. Both of the branches have a `join` continuation that refers back to the top level control frame that initiated the calling of either of the two branches.

The last control flow concept described in section 3.1 was exception handling. Like how adding exceptions to a CPS-transformed language introduced a second continuation for the closest exception handler, the solution using control frames also uses a second continuation that is chained throughout the program. This continuation labeled `ex` points to the control frame in which the closest exception handler is to be executed. In the case of the example in figure 3.7(b) this is the control frame `CF2` . The exception handler of this control frame is still the same as that of the root. A third control frame (`CF1`) is for executing the code with an updated exception handler. Therefore this control frame has its exception continuation



(a) Conditional (b) Exception handling

Figure 3.7: The control graph of the execution of an if-statement and an exception handling mechanism.

pointing at `CF2`. These three control frames do still all share the same return continuation. This is because the behavior of a return statement should be the same irregardless of whether the return was in the try or catch block.

Chapter 4

The Frame VM

In this chapter the Frame VM is introduced. This virtual machine (VM) is based on the ideas of scopes-as-frames (see chapter 2) and the notion of control frames and continuations (chapter 3).

As any VM, the Frame VM has a bytecode language, Roger, that can be used to write programs in. This chapter defines an almost complete syntax definition of this language in section 4.1. Describing a subset of the syntax reduces the number of operations greatly, resulting in a simpler VM and therefore a smaller syntax definition, less semantic rules and easier understanding. As an effect, not all valid programs can be expressed in this language subset. However, this simplified subset is chosen to still be able to express all important constructs allowed by the Frame VM. For example, expressing call/cc or exception handling mechanisms is possible but only integer addition can be used instead of all integer operations. The full definition can be found in Appendix A.

After defining the syntax, various examples of Roger programs are given. These examples expand the more generic examples from previous chapters into full example programs.

Lastly the full semantics are given of a core language of Roger (section 4.4). This core language can be derived in a number of steps that are described in subsection 4.4.2.

4.1 Roger Syntax

In this section the formal syntax of a subset of the Frame VM bytecode language Roger is introduced (see Figure 4.2). The more complex instructions are left out from this subset for simplicity. These omitted instructions are either reducible to operations in this subset of the complete language or are a trivial variation of an included instruction (subtraction and multiplication vs. addition). A list of the omitted instructions and their semantics is included in Appendix A.

A program consists of a lists of named code blocks. These blocks start with an all caps label and contain a number of ordered instructions, of which the last one is an instruction

```
1 MAIN:
2   r2 <- iload(1)
3   set(r1, [0], r2)
4   callC(get(r1, [1]), addi(iload(4), r2))
```

Figure 4.1: A code block written in Roger. From top to bottom the following syntax constructs are used: The blocks label, assigning the result of an expression to a register, execution of an instruction and the use of expression trees.

m	::= b		program
	|	$b \hookleftarrow m$	
b	::=	l**:** $\hookleftarrow i \hookleftarrow j$	block
i	::=	$i \hookleftarrow i$	instruction list
	|	$r \leftarrow e$	
	|	**set(** e **,** p **,** e **)**	
	|	**setC(** e **,** c **,** e **)**	
	|	**link(** e **,** e **,** k **)**	
	|	**link(** e **,** p **,** k **)**	
	|	**mkcurrent(** e **)**	
	|	**printc(** e **)**	
j	::=	**jump(** e **)**	control instr
	|	**jumpz(** e **,** e **,** e **)**	
	|	**callCF(** e **,** e **)**	
	|	**callC(** d **)**	
	|	**callCz(** e **,** e **,** e **,** d **)**	
d	::=	e	call args
	|	e **,** d	
q	::=	n	path element
	|	k	
	|	q **,** q	

r	::=	**r**n	register reference
p	::=	[q]	path
	|	[ ]	
e	::=	r	expression
	|	l	
	|	**nload()**	
	|	**null?()**	
	|	**iload(** z **)**	
	|	**addi(** e **,** e **)**	
	|	**eqi(** e **,** e **)**	
	|	**new(** e **)**	
	|	**new{** d **}**	
	|	**newCF(** e **)**	
	|	**newC(** e **,** e **)**	
	|	**get(** e **,** p **)**	
	|	**getC(** e **,** c **)**	
	|	**getcurrent()**	
	|	**curCF()**	
	|	**curC(** l **)**	
	|	**unpackC(** e **)**	
	|	**unpackCF(** e **)**	
	|	**rget()**	

$n \in \mathbb{N}$ $z \in \mathbb{Z}$ $l \in$ Label $c \in$ Continuation $k \in$ Link

Figure 4.2: The syntax specification of a Roger program

that influences control (e.g. making a call, returning or performing a conditional jump). An example of such a block is shown in Figure 4.1.

Besides instructions, the body also contains possibly nested expressions. The main difference between an instruction and an expression is the result of their evaluation. An expression can be seen as a function $E : S \rightarrow V, S'$. Given a state S they will evaluate to some value V an a potentially updated state. An instruction does, however, not produce a value. Instead it returns a new state. This makes that its type could be seen as $I : S \rightarrow S$. Strachey [46] and Tennent [48] have this same distinction in their work, but use different words for instruction and expression. A more detailed description of the semantics of expressions and instructions will be given in section 4.4, but it is important to remember this distinction.

As the Frame VM uses the scopes-as-frames paradigm, all memory lookups are paths in a graph representing the memory. For this reason, paths are part of the syntax and used whenever a memory operation is performed. Links in these paths are labeled by name. These names start with an ampersand (&) and a capital letter.

Like link labels, continuation labels are named as well (just like in the call graphs from chapter 3). These names start with a lowercase letter and are prefixed with the dollar sign ($).

4.2 Initial Machine State

Execution of the machine always happens with respect to three elements: The current control frame, the current data frame and the current program counter (PC). As the second is contained in the first, no active reference has to be kept to this element. With the current control frame and PC, it is possible to execute a program by stepping execution through the program. A step updates the PC and could also update the current data frame and control frame.

With this iterative execution of the machine, there has to be an initial state and a state in which the machine terminates. In the initial step, the PC is set to point at the `MAIN` block and an initial control frame is created by the machine. This control frame gets an empty data frame for its local variables and contains two continuations: One normal return address and one exception return address.

Termination is achieved when either one of these continuations is invoked. Calling the normal return address with an integer as argument, signals exiting the machine with an exitcode (where a non-zero value is used for an error). Calling the exception continuation will result in the machine terminating with an uncaught exception.

4.3 Examples

Before the formal semantics of the Frame VM and its bytecode are shown in section 4.4, it is good to have some intuition on the behavior of the machine. Therefore this section shows a number of example programs written in Roger. These example programs are derived from the examples given in chapter 2 and 3 for modeling binding and objects using data frames and modeling control flow using control frames.

The first example program is an obligatory *Hello World!*. After this first example, the construction of data objects an usage of variables is shown. Lastly programs with increasingly more complex control flow are explained. Starting at if-statements and through loops and function calls the final example will show the construction of an exception handling mechanism.

4.3.1 Hello World

```
print("Hello World!")
```

```
1  MAIN:
2          printc(iload(72))
3          printc(iload(101))
4          printc(iload(108))
5          printc(iload(108))
6          printc(iload(111))
7          printc(iload(32))
8          printc(iload(87))
9          printc(iload(111))
10         printc(iload(114))
11         printc(iload(108))
12         printc(iload(100))
13         printc(iload(33))
14
15         callC(getC(curCF(), $ret), iload(0))
```

Figure 4.3: A *Hello World!* program on the Frame VM

A *Hello World!*-program written in Roger is convoluted, but does not introduce exotic control flow or complex memory layouts. In order to print a string, each of its characters have to be printed consecutively. Getting these characters requires finding integers representing the indices in UTF-16 (in this case ASCII suffices as well). In the case of the string "Hello World!" these indices are $72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100$ and 33, We can load these integers into the machine using the `iload` operation and pass the value as an argument to the `printc` operation that prints characters.

After printing the entire string, the program must terminate in a correct way (by returning exitcode 0). This is done by getting the return handler/address of the current control frame. Getting a continuation from a control frame is done using the `getC` (get Continuation) instruction. This function requires a control frame and a label that identifies the correct continuation. In this case the label is `$ret` for the return address. We can call this continuation using `callC` (call Continuation). Providing an argument to a continuation, in this case the exitcode, is done by including its value as a second argument in the `callC` instruction. This makes the final Roger program as shown in Figure 4.3

4.3.2 Data

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

p = Point(1, 2)
print(p.x)
```

```
1  MAIN:
2          r0 <- new(iload(2))
3          set(r0, [0], iload(1))
4          set(r0, [1], iload(2))
5
6          set(getcurrent(), [0], r0)
7          r1 <- get(getcurrent(), [0, 0])
8          printc(addi(iload(48), r1))
9
10         callC(getC(curCF(), $ret), iload(0))
```

Figure 4.4: A program on the Frame VM that creates an instance of a Point and an array of integers

Using both scopes-as-types and scopes-as-frames, it is possible to model data structures using data frames. In this example (Figure 4.4) two of these data types are shown. The first is an instance of a class `Point`, the second an array of integers.

A `Point` instance is modeled as a data frame with a slot for each of the fields[1]. This data frame is created on line 2 using the `new` operation. Next, both slots of this frame/fields of this instance are populated with their values. The first slot (`x`) stores the integer 1 and the second the integer 2. To store the object in the variable `p`, the `set` operation is used on the current data frame.

Getting the value of the `x` field of the object is done by first getting the data frame from slot 0 in the current data frame, followed by getting the first slot of the resulting frame (the path $[0, 0]$ on line 7). This value can be printed to the console using the print character operation `printc`. But as this operation requires an integer representing the UTF-16 position of the character to print, we have to transform the output to the correct characters. For the values 0 to 9, the number 48 can be added to find the location of the characters 0 to 9. While this solution does not work for larger integers, it is fine for now. Lastly, the return continuation is called to terminate the VM. The integer value 0 is again given as the exitcode.

[1]Technically the class identifier and a reference to a dispatch table also require a slot, but these are left out from the example for simplicity.

4.3.3 If-statements

```
if 42 == 12:
  print(1)
else:
  print(2)
```

```
1  MAIN:
2          r0 <- eqi(iload(42), iload(12))
3          jumpz(r0, ELSE, THEN)
4
5  THEN:
6          printc(addi(iload(48), iload(1)))
7          jump(MAIN2)
8
9  ELSE:
10         printc(addi(iload(48), iload(2)))
11         jump(CONT)
12
13 MAIN2:
14         callC(getC(curCF(), $ret), iload(0))
```

Figure 4.5: An example of an if-statement in Roger

In the example from section 3.4, a conditional was modeled using a control frame for each of the branches. The Roger bytecode allows one to write the code this way by using the conditional continuation call operation `callCz`, but also allows for a much simpler construction that does not use any extra control frames. Despite that this can be seen as writing non-idiomatic code, this example uses the simpler encoding.

In this encoding a program uses the conditional jump `jumpz` instead of the conditional call. This operation checks if the first argument is equal to zero and in the case that the check returns true the label at the second position is jumped to. If the check failed, execution is jumped to the label at the second argument position.

In the example from Figure 4.5 each of these steps can be seen. First the two values are compared using the integer equality operator `eqi`. The result is compared to 0 (false) and if the check succeeds execution jumps to the `ELSE` block, otherwise the `THEN` block is invoked. These blocks print the values 1 and 2 to the console respectively. To join both branches a jump is made to `MAIN2`. This last block terminates execution by returning the exitcode 0.

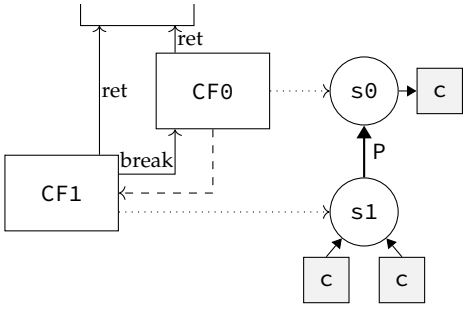
4.3.4 While Loop

One step up from a conditional on the ladder of increasingly more complicated control flow are loops. In the example shown in Figure 4.6 the compiled bytecode of a while loop can be seen. The while loop not only uses a control frame for its body, but a new scope is introduced as well for its body. For simplicity the condition is evaluated in this new scope.

Like in the previous case it is possible to model a loop using just (conditional) jumps, however this solution would not allow for the introduction of a break statement. Using a control frame, a new continuation `$break` can be added that is used for both loop termination and a potential break statement. This control frame and the new continuation are created on lines 7 and 8. The lines before this store the variable `c` in slot 0 of the current data frame and create a new data frame for the body of the loop. This new control frame is linked to the current data frame (following the empty path from the current data frame), like how the scopes are linked in the scope graph.

For creating the continuation an instruction is used that we have not seen in the examples up to this point. This operation `curC` creates a continuation of the current control frame and

```
c = 0
while c != 5:
  c += 1
```




```
1  MAIN:
2          set(getcurrent(), [0], iload(0))
3
4          r0 <- new(iload(0))
5          link(r0, [], P)
6
7          r1 <- newCF(r0)
8          setC(r1, $break, curC(MAIN2))
9          callCF(r1, WHILE_COND)
10
11 WHILE_COND:
12         r0 <- get(getcurrent(), [P, 0])
13         r1 <- eqi(r0, iload(5))
14         jumpz(r1, WHILE_BODY, WHILE_BREAK)
15
16 WHILE_BODY:
17         r0 <- get(getcurrent(), [P, 0])
18         r1 <- addi(iload(1), r0)
19         set(getcurrent(), [P, 0], r1)
20         jump(WHILE_COND)
21
22 WHILE_BREAK:
23         callC(getC(curCF(), $break))
24
25 MAIN2:
26         callC(getC(curCF(), $ret), iload(0))
```

Figure 4.6: A while-loop in a Roger program

with the provided code pointer. In this case the continuation is made to start executing the `MIAN2` block.

After setting up the new control frame for the body of the loop the `callCF` instruction is used to transfer control to the new control frame and set its PC to point to the `WHILE_COND` block. In this block the loop condition is checked. When this check succeeds execution jumps to the code block of the loop body, otherwise the block `WHILE_BREAK` is executed next.

In the loop body block the value of `c` is incremented by reading it, adding one to the value and writing the result back to the parent data frame. After this, the code jumps back to evaluating the condition. When the condition no longer holds and execution jumps to the break block, the break continuation is invoked. This transfers control back to the initial control frame and sets the PC to be at the `MIAN2` block. To terminate the VM this code block calls the return continuation with an exitcode.

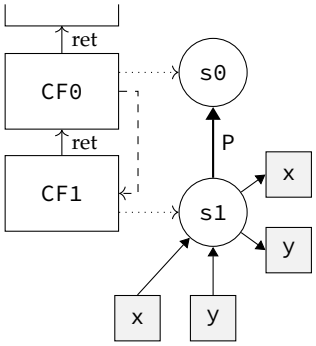
4.3.5 Function Calls

In the previous example, control was transferred to a control frame for execution of the body of a loop. Calling a function is almost exactly the same procedure. The only two differences are that the function arguments have to be stored in the function scope before calling the function and that a value is returned.

In the example Roger program from Figure 4.7 we first create the data frame for the scope of the function. This data frame has two slots, one for each function argument. Using record-syntax the argument values are directly stored in the frame. Like in the while-loop example,

```
def add(x, y):
  return x + y

print(add(1 + 2))
```




```
1  MAIN:
2      r0 <- new{iload(1), iload(2)}
3      link(r0, [], P)
4      r1 <- newCF(r0)
5      setC(r1, $ret, curC(MAIN2))
6      callCF(r1, ADD)
7
8  ADD:
9      r0 <- get(getcurrent(), [0])
10     r1 <- get(getcurrent(), [1])
11     r2 <- addi(r0, r1)
12     callC(getC(curCF(), $ret), r2)
13
14 MAIN2:
15     r1 <- rget()
16     printc(addi(iload(48), r1))
17     callC(getC(curCF(), $ret), iload(0))
```

Figure 4.7: Function calls on the Frame VM.

A program on the Frame VM that calls a function to add two numbers and prints the result.

a new control frame is created with a continuation slot set to the continuation of the current control frame at `MAIN2`. The difference is that, this time, the continuation is stored in the slot `$ret` instead of `$break`. As an effect a return handler is added to the function that returns from the function when invoked.

In the body of the function (line 9-12), the two argument values are read from its data frame. These two values are added using the integer addition operation `addi`. The result of this addition is returned to the return handler. This process is exactly the same as was used before for termination of the machine. This is because the machine is defined in such a way that returning from the initial control frame signals termination.

Instead of terminating the machine, execution now continues at the last code block. In this block the returned value is loaded from the return stack using `rget`. This value is then printed to the console and the return continuation is again invoked. But this time, we return from the initial control frame and the VM is terminated.

4.3.6 Exception Handling

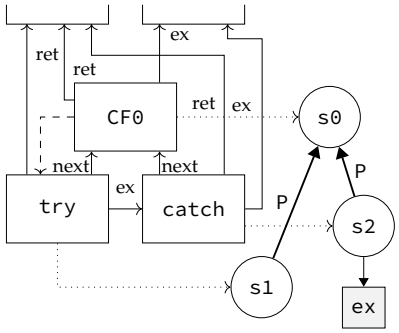
Writing a program that showcases exception handing mechanics is one of the most mind-bending programs when the Frame VM is not completely understood yet. The code example of a try-catch construction in Figure 4.8 is therefore a good test for the reader.

All concepts used should be familiar at this point, but instead of a single return address a try-catch requires three different return addresses. These correspond to the three different execution paths that can be taken inside a try or catch block. Either the block is executed completely and execution continues outside of the try-catch, an exception is thrown and execution continues in the nearest exception handler or the function that is being executed returns and execution continues at the return address of the function.

In addition to the extra continuation slots, an extra control frame is used as well. This way there is one control frame for the try block and one for the catch block[2]. The control frames for both these blocks have a new `$next` continuation that points back to the original control

[2]It is technically not required to use this second control frame, as the initial control frame could be used for executing the catch block without any problems. This is because the continuation slots of both contain identical

```
try:
  raise Exception()
except Exception as ex:
  pass
```




```
1   MAIN:
2           r0   <- new(iload(0))
3           r1 <- new(iload(1))
4           r2   <- newCF(r0)
5           r3 <- newCF(r1)
6
7           r4 <- curC(MAIN2)
8
9           setC(r2, $ex   , newC(r3, CATCH))
10          setC(r2, $next, r4)
11
12          setC(r3, $next, r4)
13
14          callCF(r2, TRY)
15
16  TRY:
17          // ...
18          callC(getC(curCF(), $ex), iload(42))
19
20  CATCH:
21          r0 <- rget()
22          set(getcurrent(), [0], r0)
23          // ...
24          callC(getC(curCF(), $next))
25
26  MAIN2:
27          callC(getC(curCF(), $ret), iload(0))
```

Figure 4.8: A program that throws and catches an exception written in Roger

frame. This continuation is invoked when the entire body of either the try or the catch block has been executed. As an effect execution continues at the instructions of the program after the try-catch block.

Returning from a function or throwing an exception in the catch block should have the same behavior as outside of the try-catch block. Therefore the exception handler and return address of the catch block should be equal to those of the original control frame. Implicit continuation copy on control frame creation takes care of this bookkeeping. As a matter of fact, the implicit copy was added for exactly this reason, as it is cumbersome to explicitly copy over unchanged continuations.

Return in the try block should also have the original behavior. Again, the implicit copy makes sure this happens automatically.

What does change is the behavior of throwing an exception in the try block. This should not invoke the exception handler of the original control frame. Instead, it should invoke the exception handler modeled by the control frame of the catch block. Therefore, the implicitly copied exception continuation needs to be overridden with a continuation of the correct exception handler.

Invoking this exception handler is done by calling its continuation with the exception value (which is in this case the integer 42). The code block of the exception handler then

values and `$next` can be eliminated when merging. But for a clear distinction between their functions, they are left as separate control frames.

pops this value from the return stack and stores it in the variable `ex`.

As a last step the try block needs to be invoked to start executing it. This is done on the last line of the `MAIN` block. After the call, the `TRY` block does some stuff and completes by throwing an exception (returning to the exception handler). The handler in the `CATCH` block does some more work and returns back to the original control frame via the `$next` continuation. This makes that normal execution is continued at `MAIN2`. Lastly the machine is terminated by calling the return handler from the initial control frame.

4.4 Semantics

In this section the formal semantics of the Frame VM are discussed. To keep the semantic rules as simple as possible, a core language of Roger is considered. In this core language, all operations that are sugar for a more primitive operation are removed and expression trees are eliminated.

4.4.1 Implementation Differences

The implementation has some slight differences from the theory described in earlier chapters. Probably the most important difference is the implicit copy of continuations. In the examples of chapter 3 bookkeeping was required to make sure all continuations of a control frame were passed to its children (like in CPS). The implicit copy does this bookkeeping automatically when a new control frame is created. As an effect a new control frame already contains all continuations from the current control frame. Therefore only the continuations that actually changed have to be updated.

A second benefit of this implicit copy is that working with control frames becomes modular. When a new continuation slot is added for a new language construct, the implicit copy makes sure all control frames get this new slot. Existing code does not have to be changed to do the bookkeeping required for the extra value and can immediately start using the new language features. An example of this modularity is shown in subsection 7.2.3, where exceptions and generator functions are added to Tiger without updating the semantics of the existing language.

Beside the implicit copy there is one other smaller difference. In the theory the PC was kept track of inside a control frame. The implementation of the Frame VM moved this to a global PC. As an effect continuations have to explicitly store the PC, but functionally nothing changed.

4.4.2 Deriving the Core Language

The core language of Roger places a number of restrictions on the syntax that can be used. This core language is a strict subset of Roger and is defined by the minimal set of instructions that an implementation of the Frame VM must support. A full syntax definition of this core language is given in Figure 4.9. As an effect all Roger programs, with the exception of programs that use debugging instructions like `debug()`, `debug!()` and `print(e)`, can be translated to this core language.

Deriving this core language is done in just two steps. The first step is to rewrite all constructs that are sugar for a set of more primitive operations. The second step is to eliminate expression trees by storing all intermediate values in registers. These two steps are described in detail in the remainder of this section.

$$
\begin{array}{lcll}
m & ::= & b & \text{program} \\
 & | & b \hookleftarrow m & \\
 & & & \\
b & ::= & l\textbf{:} \hookleftarrow i \hookleftarrow j & \text{block} \\
 & & & \\
i & ::= & i \hookleftarrow i & \text{instruction list} \\
 & | & r \leftarrow e & \\
 & | & \textbf{set}(\, r \,\textbf{,}\, p \,\textbf{,}\, r \,) & \\
 & | & \textbf{setC}(\, r \,\textbf{,}\, c \,\textbf{,}\, r \,) & \\
 & | & \textbf{link}(\, r \,\textbf{,}\, r \,\textbf{,}\, k \,) & \\
 & | & \textbf{mkcurrent}(\, r \,) & \\
 & | & \textbf{printc}(\, r \,) & \\
 & & & \\
j & ::= & \textbf{jump}(\, r \,) & \text{control instr} \\
 & | & \textbf{jumpz}(\, r \,\textbf{,}\, l \,\textbf{,}\, l \,) & \\
 & | & \textbf{callC}(\, d \,) & \\
 & & & \\
p & ::= & [\, n \,] & \text{path} \\
 & | & [\, k \,] & \\
 & | & [\,] &
\end{array}
$$

$$
\begin{array}{lcll}
r & ::= & \textbf{r}n & \text{register reference} \\
 & & & \\
d & ::= & r & \text{call args} \\
 & | & r \,\textbf{,}\, d & \\
 & & & \\
e & ::= & r & \text{expression} \\
 & | & l & \\
 & | & \textbf{nload}() & \\
 & | & \textbf{null?}() & \\
 & | & \textbf{iload}(\, z \,) & \\
 & | & \textbf{addi}(\, r \,\textbf{,}\, r \,) & \\
 & | & \textbf{eqi}(\, r \,\textbf{,}\, r \,) & \\
 & | & \textbf{new}(\, r \,) & \\
 & | & \textbf{newCF}(\, r \,) & \\
 & | & \textbf{newC}(\, r \,\textbf{,}\, r \,) & \\
 & | & \textbf{get}(\, r \,\textbf{,}\, p \,) & \\
 & | & \textbf{getC}(\, r \,\textbf{,}\, c \,) & \\
 & | & \textbf{curCF}() & \\
 & | & \textbf{unpackC}(\, r \,) & \\
 & | & \textbf{unpackCF}(\, r \,) & \\
 & | & \textbf{rget}() &
\end{array}
$$

$n \in \mathbb{N}$ $\quad z \in \mathbb{Z}$ $\quad l \in$ Label $\quad c \in$ Continuation $\quad k \in$ Link

Figure 4.9: The Roger core syntax

Desugaring Rules

When rewriting operations to a series of simpler operations, it is key to identify relations between operations. Examples of these relations that can be found in the syntax from Figure 4.2 are for instance `new{...}` and `new(...)`. These expressions both allocate a data frame on the heap, but the first also populates it with the values given where the second only allocates the data. It should be clear that there exists a set of operations which uses `new(...)` and `set(..., ..., ...)` that expresses the exact behaviour of `new{...}`.

This is exactly what can be seen in Figure 4.10. Translating the operation that allocates a data frame with record-syntax consists of two steps: First a new data frame is created with a size equal to the length of the arguments given in the record-syntax. After creating the frame, each of the record arguments are evaluated and their values are stored in the data frame using `set(.. , .., ..)` with incrementing indices.

The transformations of data frame-get and -set are a bit less straightforward. These transformations are defined recursively and in the end reduce a lookup of a path to a series of lookups with a path of length 1. In the case of a set with a path with a length longer than 1, all but the last element of the path are for finding a data frame in which to store a value. The last element in the path defines the slot in the data frame in which the value will be stored. This makes that a set of a longer path translates into a get of the first part of the path followed by a single set.

For the `get(.., ..)`-operation we can do a recursive reduction where we shorten the length of the path by 1 each iteration. This is because getting a value at a path is the same as getting the first element in the path and get the remainder of the path relative to the result of the first get. This transformation can therefore been seen as a fold from the left over the path using the `get(.., ..)`-operation as function to the fold. One thing to note here is that

f ← **new**{ $\overline{args}^{i\in[0..n]}$ }	⇒	f ← **new**($n+1$) **set**(**f,** [0], $\overline{args}^0$) ⋮ **set**(**f,** [n], $\overline{args}^n$)
set(f, [a..b, t], v)	⇒	g ← **get**(f, [a..b]) **set**(g, [t], v)
v ← **get**(f, [h | t])	⇒	g ← **get**(f, [h]) v ← **get**(g, t)
curC(lbl)	⇒	**newC**(**curCF**( ), lbl)
getcurrent( )	⇒	**unpackCF**(**curCF**( ))
link(e1, path, lbl)	⇒	**link**(e1, **get**(**getcurrent**( ), path), lbl)
callCF(e, lbl)	⇒	**callC**(**newC**(e, lbl))
callCz(e, c1, c2, a)	⇒	**jumpz**(e, THEN, ELSE) THEN: **callC**(c1, a) ELSE: **callC**(c2, a)

Figure 4.10: Desugaring rules for creating a data frame using record-syntax, getting values from frame slots, storing values in frames and calls.

there is no semantic meaning for setting a value at the empty path, where getting the empty path becomes an identity function for frame references.

Applying all the transformation rules until a fixedpoint is reached will result in a program that uses a minimal amount of different instructions. Reaching this fixedpoint allows you to continue to the second step of rewriting a program to the core language. In this second step expression trees are removed (eventhough a lot of them were introduced in this first step).

Removing Expression Trees

Removing the expression trees from a program transforms the program in such a way that the only valid subexpression is a reference to a register. To give an intuition of what this means, an example of this transformation is shown in Figure 4.11.

As unpacking the expression trees in the wrong order may result in inconsistent behaviour between the original and the transformed program, caution must be taken in applying this last translation step. In order to maintain the left to right evaluation order of subexpressions, expressions have to be unwrapped depth-first from left to right.

During this unpacking, every unpacked subexpression stores its result in a register. This register could be a new register every time, but they could also be existing registers that are no longer in use. The expression which originally contained the unpacked expression now contains a hole that should be filled with a reference to the new register.

When the depth first unpacking terminates, all subexpressions are eliminated from the program. What is left is a program in the Roger core language that can be interpreted using a minimal VM or, in this case, the formal semantics of the Frame VM.

```
BLOCK1:
        r1 <- get(getcurrent(), [P,P,0])
        set(getcurrent(), [P,P,0], addi(r1, iload(1)))
        jump(BLOCK2)
```

```
BLOCK1:
        r1 <- get(get(get(getcurrent(), [P]), [P]), [0])
        set(get(get(getcurrent(), [P]), [P]), [0], addi(r1, iload(1)))
        jump(BLOCK2)
```

```
BLOCK1:
        r7 <- getcurrent()
        r8 <- get(r7, [P])
        r9 <- get(r8, [P])
        r10 <- get(r9, [0])

        r11 <- getcurrent()
        r12 <- get(r11, [P])
        r13 <- get(r12, [P])
        r14 <- iload(1)
        r15 <- addi(r10, r14)
        set(r13, [0], r15)

        jump(BLOCK2)
```

Figure 4.11: Transforming a Roger program (top) to the core language (bottom) following the two-step process.

4.4.3 Semantic Rules

The semantic rules of Roger on the Frame VM assume that the input is correctly transformed into the core language described in the previous section. This simplifies the semantic rules and also reduces the number of rules that are needed to express all valid Roger programs.

Conventions

To improve the readability of the rules, a few notational conventions are used. In this section each of them will shortly be introduced.

First the arrows in the relations are annotated with a letter from the fraktur font (e.g. $\mathfrak{m}, \mathfrak{i}$). These are used to show the type of a relation, as all rules with a similar purpose use the same label. All rules that share the same label are grouped together and a box on the top right of each group shows the signature of the rules.

The names used for the variables in the rules are also structured to show their function and the type of data they contain. All uppercase letters denote some constant value provided at the initial step, lowercase variable names often refer to syntax constructs from Figure 4.2 and Greek letters are used for Frame VM internal values.

Examples are the letter B for a map of code blocks in a program, l for code labels and χ for the current control frame. Some variables are exceptions to this rule, like h for the heap of the VM, o for the output character stream, v for generic values and f for pointers to data

Code blocks	B	$=$	$l \rightarrow [i]$
Continuations	X	$=$	$c \rightarrow n, n \in \mathbb{N}$

Output stream	o	$=$	$[c], c \in$ UTF-16
Registers	ρ	$=$	$r \rightarrow v$
Values	v	$=$	`NullV()` | `IntV(` z `)` | `BlockPtr(` l `)` | `Frame(` f `)`
		|	`ControlFrame(` χ `)` | `Continuation(`χ, l, γ `)`
Heap	h	$=$	$x \rightarrow [\![\cdots]\!]$
Control frame	χ	$=$	x, where $h(x) = [\![f, \sigma, \rho]\!]$
Data frame	f	$=$	x, where $h(x) = [\![\lambda, \sigma]\!]$
Return stack	γ	$=$	$[v]$
Slots	σ	$=$	$n \rightarrow v$
Links	λ	$=$	$k \rightarrow f$

Figure 4.12: The notation from the syntax definition (Figure 4.2 is the base for the different letters used in the semantic rules of the Frame VM. The letters in this figure are added to this base to provide ways in which internal data or the initial state is described.

$$
\begin{array}{rlll}
D_h(f) = \sigma & \text{where} & h(f) = [\![\lambda, \sigma]\!] & \text{(data frame slots)} \\
K_h(f) = \lambda & \text{where} & h(f) = [\![\lambda, \sigma]\!] & \text{(data frame links)} \\
C_h(\chi) = \sigma & \text{where} & h(\chi) = [\![f, \sigma, \rho]\!] & \text{(control frame continuations)} \\
F_h(\chi) = f & \text{where} & h(\chi) = [\![f, \sigma, \rho]\!] & \text{(control frame data frame)} \\
R_h(\chi) = \rho & \text{where} & h(\chi) = [\![f, \sigma, \rho]\!] & \text{(control frame registers)}
\end{array}
$$

$$
\begin{array}{rll}
D_{update}(f, n, v)_{h_1} = h_2 & \text{where} & h_1[f \mapsto [\![K_{h_1}(f), D_{h_1}(f)[n \mapsto v]]\!]] = h_2 \\
K_{update}(f_1, k, f_2)_{h_1} = h_2 & \text{where} & h_1[f_1 \mapsto [\![K_{h_1}(f_1)[k \mapsto f_2], D_{h_1}(f)]\!]] = h_2 \\
C_{update}(\chi, n, v)_{h_1} = h_2 & \text{where} & h_1[\chi \mapsto [\![F_{h_1}(\chi), C_{h_1}(\chi)[n \mapsto v], R_{h_1}(\chi)]\!]] = h_2 \\
F_{update}(\chi, f)_{h_1} = h_2 & \text{where} & h_1[\chi \mapsto [\![f, C_{h_1}(\chi), R_{h_1}(\chi)]\!]] = h_2 \\
R_{update}(\chi, r, v)_{h_1} = h_2 & \text{where} & h_1[\chi \mapsto [\![F_{h_1}(\chi), C_{h_1}(\chi), R_{h_1}(\chi)[r \mapsto v]]\!]] = h_2
\end{array}
$$

$$
\begin{array}{rll}
\text{copy}_{h_1}(\chi) = \langle h_2, \chi_2 \rangle & \text{where} & \chi_2 \notin Dom(h_1),\ h_1[\chi_2 \mapsto [\![F_{h_1}(\chi), C_{h_1}(\chi), R_{h_1}(\chi)]\!]] = h_2 \\
\text{init}(n) = \sigma & \text{where} & \sigma = \text{list of } n \text{ times } \texttt{NullV()}
\end{array}
$$

Figure 4.13: Projection functions used in the semantic specification of Roger. These functions perform heap lookups and unbox the desired value from a heap allocated object.

frames on the heap. The full set of used letters is a combination of the syntax from Figure 4.2 and the notation from Figure 4.12

To simplify the notation of data frame operations, Bach Poulsen et al. [7] make use of a set of projection functions. These functions encapsulate common operations like resolving a heap pointer to a data frame to the list of slots of that frame. The semantic rules in this chapter reuse this same set of functions and add more projection functions for use with control frames. Furthermore functions are added for updating values stored in data frames and control frames. The projection and update functions are shown in Figure 4.13.

Besides the projection and update functions, Figure 4.13 also shows two other functions. The first, `copy`, takes a reference to a control frame and creates a copy of that control frame on the heap. The second initializes a list of slots to a list of null values of the given length.

Primitive Data Types

The Frame VM has a number of primitive data types that are used in the specification of the semantics. These are listed as values in Figure 4.12.

The first data type is the null value `NullV()`, the least interesting of the primitive data types. The second of the available data types is `IntV(`n`)`, an integer value. This type represents not only integers, but also booleans and characters. Booleans are represented as `IntV(0)` for false and all other values for true. Characters are represented as integers by taking their decimal UTF-16 representation. Therefore printing the integer 9786 as a character on the Frame VM results in a smiling emoji.

The third datatype is a `CodePtr(`l`)`. This is a pointer that points to the location of some executable code. Or in words, it is a label pointing to a code block in the source file.

A fourth primitive data type is `Frame(`f`)`. This datatype represents a pointer to a data frame on heap address f. This address contains a mapping from edge label names to parent data frames and an indexed list for the data slots.

The fifth datatype is the control frame (`ControlFrame(`χ`)`). Like data frames they are represented by a pointer to a heap location containing three values. The first is a pointer to a data frame, the second is a mapping of continuation labels to continuations and the third is a mapping of register names to values.

Continuations are the last primitive datatype of the Frame VM. They are represented by a reference to a control frame, a code pointer and a return stack γ.

Execution Steps

$$\boxed{X \vdash B \to \langle o, z \rangle}$$

M-Init

$$\frac{\begin{array}{c} c_1 = \texttt{Continuation(exit\_cf}, \bot\texttt{)} \qquad c_2 = \texttt{Continuation(except\_cf}, \bot\texttt{)} \\ h = [f \mapsto [\![\varnothing, \varnothing]\!], \texttt{init\_cf} \mapsto [\![f, c_1; c_2, \varnothing]\!]] \qquad \chi = \texttt{init\_cf} \\ o_1 = [\,] \qquad \gamma = [\,] \qquad l = \texttt{MAIN} \qquad l \in B \qquad B, X \vdash \langle l, h, \chi, \gamma, o_1 \rangle \xrightarrow{\mathfrak{m}} \langle o_2, z \rangle \end{array}}{X \vdash B \to \langle o_2, z \rangle}$$

Figure 4.14: Semantic rules for Frame VM initialization

The execution of Roger programs consists of three parts: machine initialization, the main loop and termination. Semantic rules for these steps are shown in Figure 4.14 and 4.15. `M-Init` is the initial rule for executing a program. This rule takes as arguments a mapping from continuation labels to indices (X) and a list of code blocks (B). With these arguments it will set up the initial machine configuration and start the main loop which, eventually[3], produces a string of characters o that forms the output of the machine and an exitcode z.

In the initial setup the initial control frame is created. This frame has an empty data frame, an empty set of registers and two continuations, one to `exit_cf` and one to `except_cf`. Invoking these continuations terminates the VM normally or with an uncaught exception respectively.

[3]Something, something, halting problem

$$\boxed{B, X \vdash \langle l, h, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{m}} \langle o, z\rangle}$$

M-StopExcept
$$\frac{\chi = \texttt{except\_cf}}{B, X \vdash \langle l, h, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{m}} \langle o, -1\rangle}$$

M-Stop
$$\frac{\chi = \texttt{exit\_cf} \qquad \gamma = [\texttt{IntV}(z)|\gamma_t]}{B, X \vdash \langle l, h, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{m}} \langle o, z\rangle}$$

M-Loop
$$\frac{\chi_1 \neq \texttt{exit\_cf} \qquad \chi_1 \neq \texttt{except\_cf} \qquad B, X \vdash \langle B(l_1), h_1, \chi_1, \gamma_1, o_1\rangle \xrightarrow{\mathfrak{b}} \langle l_2, h_2, \chi_2, \gamma_2, o_2\rangle \qquad B, X \vdash \langle l_2, h_2, \chi_2, \gamma_2, o_2\rangle \xrightarrow{\mathfrak{m}} \langle o_3, z\rangle}{B, X \vdash \langle l_1, h_1, \chi_1, \gamma_1, o_1\rangle \xrightarrow{\mathfrak{m}} \langle o_3, z\rangle}$$

Figure 4.15: Semantic rules for the Frame VM main evaluation loop and termination

The termination rules for these two cases are `M-Stop` and `M-StopExcept`. The first handles a successful termination with an integer exitcode in the return register, the second the case of an uncaught exception.

In the default case during evaluation, the current control frame χ is not equal to one of the frames captured by the rules that terminate the VM. This means that the rule `M-Loop` can be applied to evaluate the currently executing code block. The result of evaluating a code block is a new code block to execute with updated values for the heap h, the current control frame χ, return register values γ and output buffer o. These new values are fed to the rules again until the current control frame χ is one of the control frames that terminate execution.

Instruction Evaluation

$$\boxed{B, X \vdash \langle i, h, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{b}} \langle l, h, \chi, \gamma, o\rangle}$$

B-Instr
$$\frac{B, X \vdash \langle i, \chi_1, \gamma_1, h_1, o_1\rangle \xrightarrow{\mathfrak{i}} \langle h_2, \gamma_2, o_2\rangle \qquad B, X \vdash \langle is, h_2, \chi_1, \gamma_2, o_2\rangle \xrightarrow{\mathfrak{b}} \langle l_2, h_3, \chi_2, \gamma_3, o_3\rangle}{B, X \vdash \langle [i|is], h_1, \chi_1, \gamma_1, o_1\rangle \xrightarrow{\mathfrak{b}} \langle l_2, h_3, \chi_2, \gamma_3, o_3\rangle}$$

B-Control
$$\frac{B, X \vdash \langle i, h_1, \chi_1\rangle \xrightarrow{\mathfrak{c}} \langle l, h_2, \chi_2, \gamma_2\rangle}{B, X \vdash \langle [i], h_1, \chi_1, \gamma_1, o_1\rangle \xrightarrow{\mathfrak{b}} \langle l, h_2, \chi_2, \gamma_2, o_1\rangle}$$

Figure 4.16: Semantic rules for evaluation of code blocks

In the semantic rules code blocks are represented by a list of instructions i. Evaluating a block uses the two rules in Figure 4.16. These two rules fold over the list of instructions, where the second rule evaluates the last instruction (which updates the control) and the first evaluates all normal instructions. Rules describing the semantics of these two types of instructions are shown in Figure 4.17 and 4.18 respectively.

The evaluation rules for non-control instructions all influence the state of the VM without updating the PC or current control frame. In the case of `I-PrintChar`, updating the state only means to append a single character to the output stream of the runtime. For this its argument

$$\boxed{B, X \vdash \langle i, h, \chi, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h, \gamma, o\rangle}$$

I-PrintChar
$$\frac{R_h(\chi)(r) = \texttt{IntV}(n)}{B, X \vdash \langle \textbf{printc}(r), h, \chi, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h, \gamma, o; \texttt{chr}(n)\rangle}$$

I-MkCurrent
$$\frac{R_{h_1}(\chi)(r) = \texttt{Frame}(f) \qquad F_{update}(\chi, f)_{h_1} = h_2}{B, X \vdash \langle \textbf{mkcurrent}(r), h_1, \chi, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h_2, \gamma, o\rangle}$$

I-Link
$$\frac{R_{h_1}(\chi) = \rho \qquad \rho(r_1) = \texttt{Frame}(f_1) \qquad \rho(r_2) = \texttt{Frame}(f_2) \qquad K_{update}(f_1, k, f_2)_{h_1} = h_2}{B, X \vdash \langle \textbf{link}(r_1, r_2, k), h_1, \chi, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h_2, \gamma, o\rangle}$$

I-SetContinuation
$$\frac{\begin{array}{c} R_{h_1}(\chi_1) = \rho \qquad \rho(r_1) = \texttt{ControlFrame}(\chi_2) \\ \rho(r_2) = k \qquad k = \texttt{Continuation}(\chi_2, l) \qquad C_{update}(\chi_1, X(c), k)_{h_1} = h_2 \end{array}}{B, X \vdash \langle \textbf{setC}(r_1, c, r_2), h_1, \chi_1, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h_2, \gamma, o\rangle}$$

I-Set
$$\frac{\begin{array}{c} R_{h_1}(\chi) = \rho \\ \rho(r_1) = \texttt{Frame}(f) \qquad \rho(r_2) = v \qquad n \in Dom(D_{h_1}(\chi)) \qquad D_{update}(\chi, n, v)_{h_1} = h_2 \end{array}}{B, X \vdash \langle \textbf{set}(r_1, [n], r_2), h_1, \chi, \gamma, o\rangle \xrightarrow{\mathrm{i}} \langle h_2, \gamma, o\rangle}$$

I-Assign
$$\frac{B, X \vdash \langle e, R_{h_1}(\chi), \chi, \gamma_1, h_1\rangle \xrightarrow{\mathfrak{e}} \langle v, h_2, \gamma_2\rangle \qquad R_{update}(\chi, r, v)_{h_2} = h_3}{B, X \vdash \langle r \leftarrow e, h_1, \chi, \gamma_1, o\rangle \xrightarrow{\mathrm{i}} \langle h_3, \gamma_2, o\rangle}$$

Figure 4.17: Semantic rules for evaluating instructions

is evaluated to an integer n. Using the function `chr` this integer is converted to an UTF-16 character that has the decimal encoding equal to n.

Changing the current data frame is handled by the `I-MkCurrent` rule. First the argument is evaluated to a frame reference f. To update the current frame, the heap value of the current control frame χ is updated to be the new data frame f, the old continuation slots and the old register values.

Linking a data frame to a second data frame using a link label k is shown in the `I-Link` rule. In this rule, both frames are evaluated to the pointers f_1 and f_2. Using the update notation, the map of links of f_1 is then updated to make link label k map to f_2.

Storing a continuation in a control frame follows almost the same steps. Rule `I-SetContinuation` first evaluates the control frame and the continuation. Then the list of continuations of the control frame χ_2 is updated to store the new continuation at the index specified by the continuation label c in X.

Storing a value in a slot of a data frame is, again, a similar procedure. Instead of updating the links when linking two frames, storing a value updates the list of slots of the data frame. The index to store at in this list is no longer provided by a global mapping like X but instead provided as an argument.

The last instruction described by the semantic rules for instructions is assigning to a reg-

$$\boxed{B \vdash \langle j, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{c}} \langle \chi, l, h, \gamma\rangle}$$

C-Jump
$$\frac{R_h(\chi)(r) = \texttt{BlockPtr}(l) \qquad l \in B}{B \vdash \langle \textbf{jump}(r), h, \chi, \gamma\rangle \xrightarrow{\mathfrak{c}} \langle \chi, l, h, \gamma\rangle}$$

C-JumpFalse
$$\frac{\begin{array}{c} R_h(\chi) = \rho \\ \rho(r_1) = \texttt{IntV}(n) \qquad n \neq 0 \qquad \rho(r_2) = \texttt{BlockPtr}(l_1) \qquad \rho(r_3) = \texttt{BlockPtr}(l_2) \end{array}}{B \vdash \langle \textbf{jumpz}(r_1, r_2, r_3), h, \chi, \gamma\rangle \xrightarrow{\mathfrak{c}} \langle \chi, l_2, h, \gamma\rangle}$$

C-JumpTrue
$$\frac{\begin{array}{c} R_h(\chi) = \rho \\ \rho(r_1) = \texttt{IntV}(n) \qquad n = 0 \qquad \rho(r_2) = \texttt{BlockPtr}(l_1) \qquad \rho(r_3) = \texttt{BlockPtr}(l_2) \end{array}}{B \vdash \langle \textbf{jumpz}(r_1, r_2, r_3), h, \chi, \gamma\rangle \xrightarrow{\mathfrak{c}} \langle \chi, l_1, h, \gamma\rangle}$$

C-CallC
$$\frac{\begin{array}{c} R_{h_1}(\chi_1) = \rho \\ \rho(r) = \texttt{Continuation}(\chi_2, l, \gamma_2) \qquad \text{copy}_{h_1}(\chi_2) = \langle h_2, \chi_3\rangle \qquad \langle rs, \rho, \gamma_2\rangle \xrightarrow{\mathfrak{el}} \gamma_3 \end{array}}{B \vdash \langle \textbf{callC}([r|rs]), h_1, \chi_1, \gamma_1\rangle \xrightarrow{\mathfrak{c}} \langle \chi_3, l, h_2, \gamma_3\rangle}$$

$$\boxed{\langle [r], \rho, \gamma\rangle \xrightarrow{\mathfrak{el}} \gamma}$$

H-ExpListEmpty
$$\frac{}{\langle [], \rho, \gamma\rangle \xrightarrow{\mathfrak{el}} \gamma}$$

H-ExpList
$$\frac{\rho(r) \to v \qquad \gamma_2 = [v|\gamma_1] \qquad \langle rs, \rho, \gamma_2\rangle \xrightarrow{\mathfrak{el}} \gamma_3}{\langle [r|rs], \rho, \gamma_1\rangle \xrightarrow{\mathfrak{el}} \gamma_3}$$

Figure 4.18: Semantic rules for evaluating control-influencing instructions

ister (`I-Assign`). This instruction evaluates the expression e to some value v and updates the registers ρ to store this value.

The rules for evaluating control influencing instructions (Figure 4.18) are in general a bit simpler as no heap updates are required. Of these rules only `C-CallC` does something more complex than updating the current control frame χ or the program counter (PC) l.

In the case of a (conditional) jump, the register reference is looked up in the register values $R_h(\chi)$ to get a code pointer b. This value is checked against the set of available code blocks B and if valid, returned as the new code block to execute. For the conditional jump, there is one extra step, as it must be decided which branch should be executed. This is done by comparing the integer value n to the number 0.

The rule for calling a continuation (`C-CallC`) is more complex. In part, this is because the control frame stored in the continuation cannot be used directly. This would mutate the state stored in the continuation which is not desired when continuations are used multiple times. A solution to this is to first create a copy of the control frame. Leaving out this copy results in an implementation of one-shot-continuations, continuations that can be invoked only once.

The second part that makes this rule more complex is that it takes a variable amount of

arguments that will be put on the return stack. For this the helper relation $\xrightarrow{\text{cf}}$ is used. These rules evaluate each of the arguments one by one and pushes the results on the stack of return values γ.

Expression Evaluation

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma \rangle \xrightarrow{\text{e}} \langle v, h, \gamma \rangle}$$

E-Label
$$\frac{l \in B}{B, X \vdash \langle l, \rho, h, \chi, \gamma \rangle \xrightarrow{\text{e}} \langle \texttt{BlockPtr}(l), h, \gamma \rangle}$$

E-LoadInt
$$\frac{}{B, X \vdash \langle \textbf{iload}(n), \rho, h, \chi, \gamma \rangle \xrightarrow{\text{e}} \langle \texttt{IntV}(n), h, \gamma \rangle}$$

E-LoadNull
$$\frac{}{B, X \vdash \langle \textbf{nload}(), \rho, h, \chi, \gamma \rangle \xrightarrow{\text{e}} \langle \texttt{NullV}(), h, \gamma \rangle}$$

E-AddInt
$$\frac{\rho(r_1) = \texttt{IntV}(v_1) \qquad \rho(r_2) = \texttt{IntV}(v_2) \qquad v_1 + v_2 = v}{B, X \vdash \langle \textbf{addi}(r_1, r_2), \rho, h, \chi, \gamma \rangle \xrightarrow{\text{e}} \langle \texttt{IntV}(v), h, \gamma \rangle}$$

Figure 4.19: Semantic rules for creating basic primitive values and performing integer addition.

When evaluating the instructions there was one rule that needed to evaluate a sub-expression. This rule for the assignment of register values used the relation $\xrightarrow{\text{e}}$ to this end. Expressions evaluated this way do not alter the state of the VM besides allocating data/control frames on the heap and requesting returned values, which make the rules relatively simple.

Figure 4.19 shows the semantic rules that create the primitive values `IntV(_)`, `NullV()` and `BlockPtr(_)` (`E-LoadInt`, `E-LoadNull` and `E-Label` respectively). The first three rules in this figure do nothing unexpected and just produce primitive values. The last rule in this figure (`E-AddInt`) does some more operations to perform integer addition. This addition consists of looking up the values in the registers of the current control frame, verifying them being integers, adding the two values together and creating a new primitive integer value containing the result of the summation.

A different type of operation on integers is an operation that compares two integers. Integer comparison has two rules, one for the case when the integers are equal and one for when they are not. These rules are shown in Figure 4.20. Depending on the result of the comparison the rules will either evaluate to true (`IntV(1)`) or false (`IntV(0)`). The other two rules in Figure 4.20 are for performing a null check. This null check either evaluates its argument to an instance of `NullV()`, resulting in true, or to some other value, in which case the result is false.

In addition to rules for creating primitive values and simple checks, the Frame VM has a number of expressions related to creating data frames, control frames and continuations. The semantic rules for these expressions are shown in Figure 4.21.

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma \rangle}$$

E-IsNull-True
$$\frac{\rho(r) = \texttt{NullV()}}{B, X \vdash \langle \textbf{null?}(r), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{IntV(1)}, h, \gamma \rangle}$$

E-IsNull-False
$$\frac{\rho(r) \neq \texttt{NullV()}}{B, X \vdash \langle \textbf{null?}(r), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{IntV(0)}, h, \gamma \rangle}$$

E-EqualInt-True
$$\frac{\rho(r_1) = \texttt{IntV}(v_1) \qquad \rho(r_2) = \texttt{IntV}(v_2) \qquad v_1 = v_2}{B, X \vdash \langle \textbf{eqi}(r_1, r_2), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{IntV(1)}, h, \gamma \rangle}$$

E-EqualInt-False
$$\frac{\rho(r_1) = \texttt{IntV}(v_1) \qquad \rho(r_2) = \texttt{IntV}(v_2) \qquad v_1 \neq v_2}{B, X \vdash \langle \textbf{eqi}(r_1, r_2), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{IntV(0)}, h, \gamma \rangle}$$

Figure 4.20: Semantic rules for comparing integers and checking the type of a primitive value

The first rule `E-New` allocates a new data frame on the heap with a given size. This size is gotten by evaluating its argument to an integer. A new list of slots σ is created with this size using the `init` function. Together with a map for all the frame links λ (which is still empty at this point in time), this defines the data frame $[\![\lambda, \sigma]\!]$. Lastly the heap h is updated in such a way that a fresh pointer x points to the new data frame.

Creating a new control frame is closely related to creating a new data frame. The difference is that the list of links is replaced by a pointer f to a data frame and the list of slots is not initialized with null-pointers but by the continuations of the current control frame. This copy

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma \rangle}$$

E-New
$$\frac{\rho(e) = \texttt{IntV}(n) \qquad x \notin Dom(h) \qquad \text{init}(n) = \sigma}{B, X \vdash \langle \textbf{new}(e), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{Frame}(x), h[x \mapsto [\![\varnothing, \sigma]\!]], \gamma \rangle}$$

E-NewCF
$$\frac{\rho(e) = \texttt{Frame}(f) \qquad x \notin Dom(h) \qquad C_h(\chi) = \sigma}{B, X \vdash \langle \textbf{newCF}(e), \rho, h, \chi, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{ControlFrame}(x), h[x \mapsto [\![f, \sigma, \varnothing]\!]], \gamma \rangle}$$

E-NewContinuation
$$\frac{\rho(e_1) = \texttt{ControlFrame}(\chi_1) \qquad \rho(e_2) = \texttt{BlockPtr}(l) \qquad \text{copy}_{h1}(\chi_1) = \langle h_2, \chi_2 \rangle}{B, X \vdash \langle \textbf{newC}(e_1, e_2), \rho, h_1, \chi_1, \gamma \rangle \xrightarrow{\mathfrak{e}} \langle \texttt{Continuation}(\chi_2, l, \gamma), h_2, \gamma \rangle}$$

Figure 4.21: Semantic rules for expressions that create data frames, control frames and continuations

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-GetSlot
$$\frac{\rho(r_1) = \texttt{Frame}(f) \qquad D_h(f)[n] = v}{B, X \vdash \langle \textbf{get}(r_1, [n]), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-GetLink
$$\frac{\rho(r) = \texttt{Frame}(f) \qquad K_h(f)(k) = v}{B, X \vdash \langle \textbf{get}(r, [k]), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-GetEmpty
$$\frac{\rho(r) = \texttt{Frame}(f)}{B, X \vdash \langle \textbf{get}(r, []), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle \texttt{Frame}(f), h, \gamma\rangle}$$

Figure 4.22: Semantic rules for expressions that operate on data frames.

of continuations on control frame creation is called the implicit (control frame) copy throughout this thesis. This implicit copy enables modular construction of control flow. Lastly, an initially empty map is added to the control frame that contains register values.

The creation of a continuation object is different in that it does not allocate values on the heap. This is not needed, as continuation objects, in contrast to control frames and data frames, are not mutable. A continuation has three components: A copy of the current control frame, the provided code pointer and the current return stack. With this information the execution state can be correctly restored when a continuation is invoked.

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-GetContinuation
$$\frac{\rho(r) = \texttt{ControlFrame}(\chi_2) \qquad C_h(\chi_2)[X(c)] = v}{B, X \vdash \langle \textbf{getC}(r, c), \rho, h, \chi_1, \gamma\rangle \rightarrow \langle v, h, \gamma\rangle}$$

E-CurCF
$$\frac{}{B, X \vdash \langle \textbf{curCF}(), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle \texttt{ControlFrame}(\chi), h, \gamma\rangle}$$

E-unpackContinuation
$$\frac{\rho(r) \rightarrow \texttt{Continuation}(\chi_2, l)}{B, X \vdash \langle \textbf{unpackC}(r), \rho, h, \chi_1, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle \texttt{ControlFrame}(\chi_2), h, \gamma\rangle}$$

E-unpackCF
$$\frac{\rho(r) \rightarrow \texttt{ControlFrame}(\chi_2) \qquad F_h(\chi_2) = f}{B, X \vdash \langle \textbf{unpackCF}(r), \rho, h, \chi_1, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle \texttt{Frame}(f), h, \gamma\rangle}$$

E-ResultGet
$$\frac{\gamma_1 = [v|\gamma_2]}{B, X \vdash \langle \textbf{rget}(), \rho, h, \chi, \gamma_1\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma_2\rangle}$$

Figure 4.23: Semantic rules for miscellaneous expressions.

By applying the transformation rules to transform a Roger program into the core language, it is made sure that frame operations only operate on paths containing a single step. This makes that the get operation only needs two rules in the semantics: One for the case where the path element is a slot index (`E-GetSlot`) and one for when it is a link label (`E-GetLink`). In both cases the argument is evaluated to a frame pointer, after which either a link is looked up from the set of links or an indexed slot is resolved from the list of slots.

A get of an empty path is a third valid variant of the get function. This operation returns the frame that is found by traversing the empty path from the given frame. In practice this means that it returns the argument value if it is a pointer to a data frame. Therefore the `E-GetEmpty` rule only needs to validate that the argument is indeed a frame before returning it.

The last figure containing semantic rules for expressions is Figure 4.23. Getting a continuation value from a control frame is the first rule in this figure. This rule evaluates its first argument to a control frame and performs a lookup in its list of continuations $C_h(\chi_2)$. The index for this lookup is found by resolving the continuation label c to an index using the continuation map X.

The second rule is `E-CurCF`, which is used for getting the current control frame. This value is already provided as χ and this operation therefore becomes a boxing operation for this value.

The next two semantic rules are for unpacking a control frame or a data frame from a continuation and control frame respectively. In the case of unpacking a continuation it is enough to evaluate the register to a continuation value. This is because this continuation value already contains the control frame that needs to be unpacked. Unpacking a control frame to get its data frame requires one extra step as application of a projection rule is required to get the frame reference from the heap.

The last rule (`E-ResultGet`) pops a value from the return register. This register γ can be accessed directly, making it easy to split it in a head and a potentially empty tail. The head v then becomes the result of the operation and the updated return register is added to this result.

Chapter 5

Language Engineering Project

The Language Engineering Project (LEP) is a master level course on the TU Delft[1]. In this course students implement a programming language in the Spoofax language workbench. The goal of this course is twofold. On the one hand, the students get to understand challenges faced in language design. On the other hand, the students explore boundaries of the workbench and test applications of new Domain Specific Languages (DSLs) for language specifications.

All projects in the course have a different focus. Some are aimed at static analysis of specific language constructs (e.g. borrow checking in Rust), others perform a case study of implementing a full programming language. For two projects the focus was to compile a programming language to the Frame VM. One took Prolog as the source language, the other Rust. Both these projects can be seen as good case studies for the Frame VM. Not only for the applicability of the constructs provided by the VM and for finding missing features, but also for testing the usability of the system.

In the first sections of this chapter some interesting aspects of these two compilers are shown. For Prolog this is the backtracking and for Rust the object model. More interesting to this thesis are however the implications the project has had for the development of the Frame VM. These improvements and changes are discussed in the later sections together with a retrospect on the usability of the Frame VM.

5.1 Compiling Prolog

Prolog is a declarative logic programming language that makes use of backtracking and unification in its execution. This means that the control flow of a program is not directly clear from the abstract syntax tree (AST) and more complex than the patterns described in section 3.1. As an effect, Prolog was an interesting case study in how to implement more declarative languages and backtracking on the Frame VM.

To compile backtracking, the implementation uses mechanisms like generator functions and try-catch mechanisms. A full description of the working of this system is given in the original report by Miljak [33], but the Python code in Figure 5.1 gives an intuition for the implemented system. In short, execution of a procedure results in a list of all potential options that can be chosen during execution. Each rule in the body of this procedure can introduce these options. This makes that the total options of a procedure are an aggregation of all options for the rules in a procedure. Using Python-like generator functions, these options can be yielded creating a depth first search over all options. When a path fails, execution will continue to the next yield point, implementing efficient backtracking.

[1] https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=45606

```
def procedure(arg1, arg2, arg3):
    for e in rule1(arg1, arg2, arg3):
        yield e
    for e in rule2(arg1, arg2, arg3):
        yield e
```

Figure 5.1: Pseudocode of execution of a Prolog rule with three arguments and two rules with three arguments each [33].

Implementing the control flow that simulates a generator function was found to be elegant to do on the Frame VM. It was however hard to write a proper compiler for Prolog, as it is really easy to end up with an interpreter. Furthermore compiling a language as dynamic as Prolog to a non-dynamic target language was not straightforward.

5.2 Compiling Rust

Rust is a much more conventional language when comparing it to Prolog. There are however a lot of lessons to be learned from writing a compiler for such a language.

For example functions, and especially nested functions, posed a problem when compiling Rust. In short, unique labels had to be generated for the code generated by the function bodies to uniquely identify them. This problem was fixed by pre-generating unique identifiers for each function. Furthermore, the blocks created by the compiled functions interfered with the code generated at the call site. This problem was worked around by inserting extra `jump` statements. While this created a lot of blocks to which only a single jump referenced (and thus should have been inlined), this did resolve the issue. The full report by Crielaard and Beinema provides more details about these problems and their solution.

A more interesting case was how internal data was encoded. Encoding integers on the Frame VM was trivial, as integers are already a primitive datatype. Strings were not a primitive datatype[2], so some extra steps were required. It was decided to encode a string as a data frame with the first slot containing the length of the string and one slot for a each of the characters in the string. An example of an encoded string can be seen on the right side of Figure 5.2.

For encoding enums and objects, data frames were used as well. Now the first slot contained a type identifier and each of the fields of the objects were stored in separate slots. A

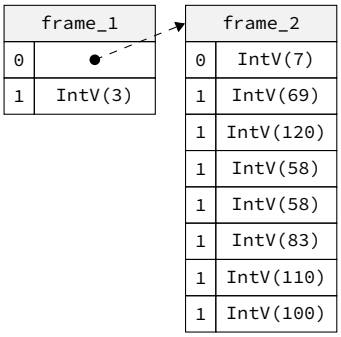


Figure 5.2: Encoding of an enum instance `Ex:Snd` on the Frame VM [11].

[2]There was also no functionality in the Frame VM for working with single characters at the start of the Rust project. Support for them was added when work started on the Rust string implementation.

```
from stdlib.strings import concat

BLOCK:
        // assume r0 and r1 contain frames representing strings
        r2 <- new{r0, r1}
        r3 <- newCF(r2)
        setC(r3, $ret, curC(BLOCK2))
        callCF(r3, concat)

BLOCK2:
        r4 <- rget()
        // r4 now contains the concatenated string
```

Figure 5.3: The syntax of using imported functions in Roger. This example concatenates two strings using the `concat` function from `stdlib.strings`.

string with the full path of the enum was decided as the type identifier. While this was more memory intensive than using consecutive integers as identifiers, it was more practical during debugging. As a result an instance `Snd`, containing a single integer, of an enum `Ex` is encoded in the way shown in Figure 5.2. The identifier points to a second data frame storing the string `Ex::Snd` and the integer value is stored in the second slot of the data frame.

5.3 Frame VM Improvements

The creation of the Rust and Prolog compilers has had a big influence on the development of the Frame VM. This is in part because a solution in a production environment has different design goals than a research project. As an effect certain features were missing or incomplete in the bytecode language and in the utility functions that aided in the code generation.

In a research implementation it would for example have sufficed to have a print function that prints raw internal data to the console (e.g. `IntV(42)`). In a practical compiler such an instruction would not suffice however. The output of a real program would need to consist of a string of characters, in which integers, booleans, objects and actual strings could be represented. To this end an instruction was added that prints single characters to the console and the original print instruction was labeled as a *debug* instruction.

5.3.1 Standard Library

In order to make effective use of these character operations, various functions needed to be created. Examples of these are printing a string, concatenating a string, converting an integer to a string and printing an integer. Providing a standard library that would contain these functions, would be useful to compiler developers. This way they do not have to write these function by themselves with the chance that they would introduce bugs when doing so.

During the course of the Language Engineering Project, such a library was implemented for the Frame VM. The syntax for the imports is heavily inspired by Python in that it uses the keywords `import` and `from`. An example of a program that concatenates two strings using imported functions is shown in Figure 5.3.

A number of library functions were implemented and included in this standard library. Most of them were related to strings, but a function for comparing data was also created.

Some of these functions were provided by the students as they created them, others were developed as part of this thesis.

5.3.2 Empty Slots

Besides the addition of the previously mentioned instruction for outputting characters, a second group of instructions was also missing from the language. These instructions were all related to slots of a data frame being empty. At that time, getting a value from an empty slot would crash the VM. Normally this is not an issue as references always happen after assignment of a (default) value. This does however become an issue when frames have to be copied, or in the case of Rusts match expressions, compared. Now the program must be able to skip the empty slots to prevent a crash, but it has no way of knowing whether a slot is empty.

Initially this was solved by adding instructions that could check if a slot was empty and instructions that could explicitly set a slot to empty. Later on, these instructions were removed and replaced by an instruction for performing null checks and a change to the behavior of loading a value from an empty slot. This changed behavior meant that getting an empty slot would now produce a null value instead of crashing the machine.

This change also solved an issue that arose in the Tiger compiler. Here null values were initially represented by empty data frames. However, these different null values were actually different values and could therefore not effectively be compared.

5.3.3 Code Generation

While functions where provided that aided in code generation, code generation was still relatively tedious. For one, the compilers had to work with abstract syntax instead of concrete syntax. Furthermore, a lot of boilerplate code was needed to produce a working program. Most of the concepts in this boilerplate were found in all compilers written to the Frame VM. Providing an environment in which this boilerplate was abstracted away would therefore reduce the workload required for creating a compiler greatly.

It was also observed that a lot of inefficient code was generated by the compilers. An example of this is that extra data frames need to be created in order to maintain the correspondence to the scope graph. These frames are often not needed at runtime and can therefore be optimized away.

While the Rust compiler implemented various bytecode optimizations, it would be better to integrate these optimizations in the VM itself. This would mean that much of the bytecode overhead produced by code generation could automatically be removed after compilation. As an effect, the compiler implementer only has to implement language specific optimizations.

As creating a good system for bytecode optimizations is hard, these optimizations were not implemented and left as future work (see subsection 9.2.1). In fact, investigating bytecode optimizations is likely a very interesting master thesis project in itself.

5.3.4 Testing the Compilers

A large bottleneck in the development of the two compilers was the lack of a good way to test the compilers. Due to the inclusion of library code, the generated bytecode needed to reference the imported files at runtime. Enabling the multi-file analysis mode that this required limited the compatibility with the SPT testing framework included in Spoofax. Therefore it was not possible to run automated test suites, making it impossible to write test cases for a compiler to the Frame VM. The Rust project investigated a combination of Sunshine and Nailgun that might be able to work around this issue [11], but this solution did not work and both projects had to use manual testing.

A while after both projects concluded a solution to this problem was found in pre-compiling library files. Importing these libraries at runtime can then be performed by loading the files from the disk. While this is a less elegant solution, this means that it is now possible to run programs without performing the analysis steps. With this implemented, a helper strategy was constructed that interfaced between the SPT testing framework and the execution on the VM, allowing for automated testing. (see section 6.4).

5.4 Frame VM Applicability

The two projects described in this chapter showed that it was possible to create compilers for various programming languages that varied greatly in their internals. Before the project started, only a rudimentary compiler for a subset of Scheme was constructed for testing purposes and a lot of toy-programs were written by hand that exercised various control flow constructions. By compiling these different types of languages various shortcomings of the design were discovered. Most of these were quickly fixed by either extending the bytecode instructions or fixing behavioral bugs of either the machine itself or of the helper functions aiding in the compilation steps.

It was found that the memory model was sufficient to encode different values like integers, strings, enums and objects [11]. Furthermore the model of control frames and their multiple return addresses as continuations proved to allow for elegant encoding of backtracking [33].

Compiling high level concepts like case-matching and unification proved to be hard as the target level was very low-level. The shorthands provided in the bytecode for more complex operations like function calls and yields helped, but were not enough to alleviate this problem. This was to be expected as all other operations remained low-level and a function call is not that high-level anyway.

It was also clear that writing a compiler to the Frame VM required a lot of boilerplate code. Big chunks of such boilerplate code was present in both the compilers (and the Scheme compiler) in some shape or form. The helper function that could construct a valid AST from a flattened representation simplified the code generation quite a lot, but keeping track of created blocks remained a tough task.

Near the end of the Language Engineering Project (LEP), a start was made with designing a DSL that would make writing compilers to the Frame VM a lot more streamlined. This design took note of most of the issues encountered during the project and tried to provide solutions for them. The DSL became what is now called Dynamix, and is described in complete details in the next chapter.

Chapter 6

Dynamix

The last chapter showed that it is possible to write compilers from various languages to the Frame VM. While the VM allowed for elegant solutions with respect to control flow and had enough coverage to execute many language components, the process of code generation itself could be improved greatly. The two compilers from the Language Engineering Project (LEP) and a Scheme compiler developed during this thesis all had the problem that lots of boilerplate code was needed.

An effect of this was that a lot of knowledge about code generation and the Frame VM was required to generate correct code. By using a Domain Specific Language (DSL) most of this required knowledge can be abstracted away into a specification of the semantics of a language. This specification can then be used to generate a compiler without requiring any knowledge about the compiler internals.

An existing DSL for writing the semantics of a language is Dynsem [50]. This language uses a meta-interpreter that interprets both the semantics of a language and a source program to evaluate to a value. Dynsem is already tightly integrated into the Spoofax language workbench and has library features that allow for semantics written using the scopes-as-frames paradigm [51]. Initially the plan was to write a second Dynsem back end for meta-compilation instead of meta-interpretation. But when the idea of control frames materialized for modeling execution of control flow constructs it was clear that Dynsem had no good mechanisms to describe the control flow of a program. It suffered from the same problems as many semantics frameworks, in that describing control flow of a program required describing the control flow throughout all semantic rules, even those not concerned with the control flow of a program.

Therefore it was decided that a new declarative DSL was needed for describing the semantics of a language. This new language had to be designed with describing the control flow of a program and scopes-as-frames in mind. In this chapter this new language, Dynamix, is introduced by constructing a declarative description of the semantics of Tiger. This description can then be used by the Dynamix meta-compiler to compile a subset of all Tiger programs to Frame VM bytecode.

In addition to the example specification, the full syntax and semantics of Dynamix are given in section 6.1 and 6.5 respectively.

6.1 Syntax

A Dynamix program is built around a list of rules. When a rule is applied its arguments are matched with the given parameters producing a set of bindings. These are then used to evaluate the body of the rule. In this body other rules and primitive operations can be called and values can be assigned to variables. The final result of evaluating the body of a rule is a state transformation which optionally produces a value.

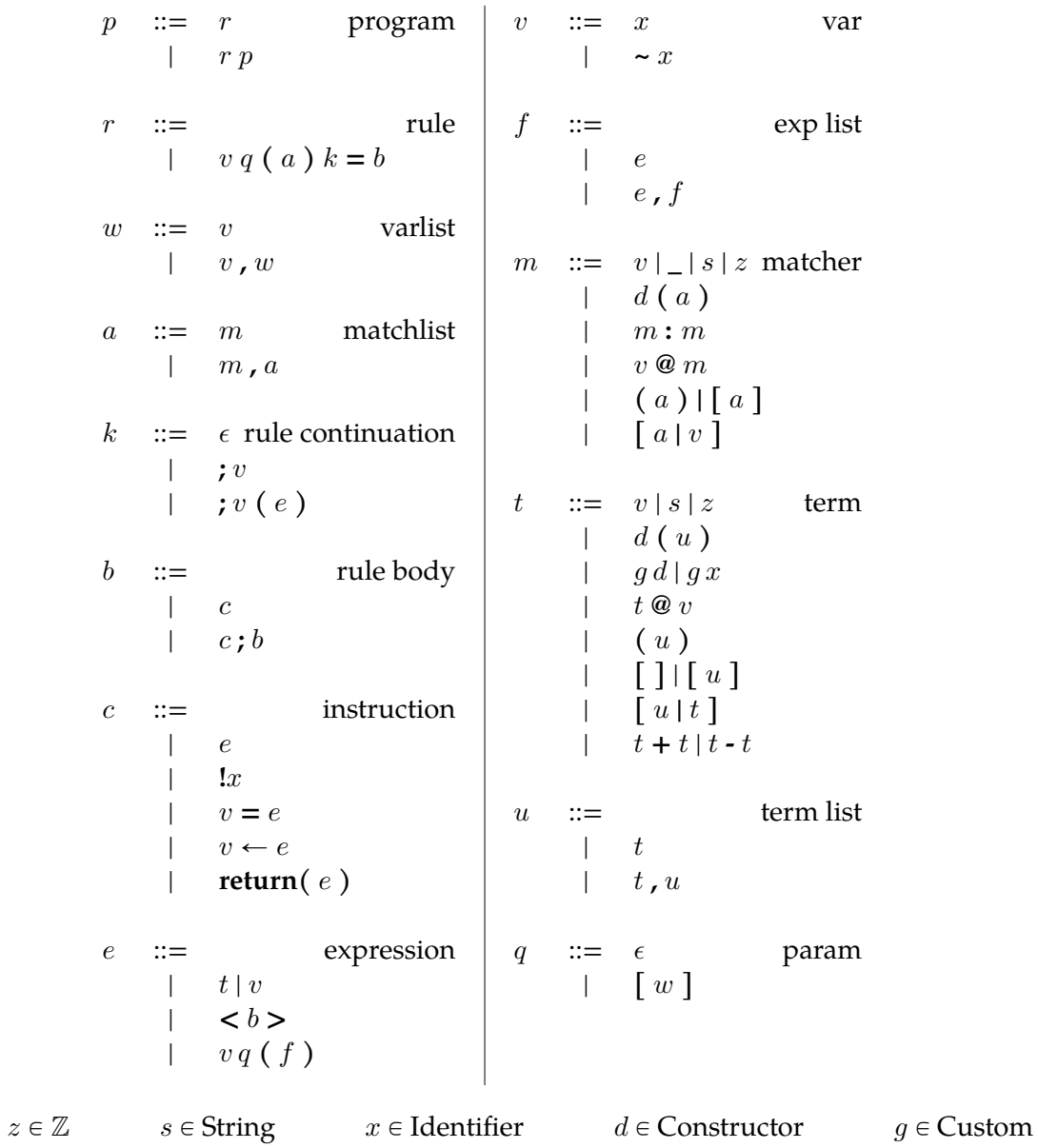


Figure 6.1: Syntax specification of Dynamix

This process is reflected in the syntax of the language (Figure 6.1): A full specification is a collection of rules, a rule has a matcher and a body and the body contains calls to other rules and construction of terms. These terms use constructors that start with a capital letter and can contain both numbers, normal characters and underscores. Identifiers follow a similar pattern, but start with a lowercase letter. Custom constructors consist of a combination of special characters (that are not used anywhere else in the syntax).

Certain components in this syntax have additional syntax highlighting to show their purpose at a glance. Names of rules have a blue color, Frame VM primitive operations are orange and Dynamix primitives are dark green. In addition to these names used in calls, constructors of terms are colored dark teal.

Dynamix is designed in such a way that it can use multiple back-ends. This has the effect that there is no notion of the Frame VM bytecode-language Roger in the syntax of Dynamix. Instead, primitive operations are described in a generic way: They take some arguments and produce AST elements. Primitive operations are therefore described as a separate mapping

```
int       →  iload
null      →  nload
char      →  cload

jump      →  jump
jumpz     →  jumpz
callC     →  callC
callCF    →  callCF
curCF     →  curCF
curC      →  curC
newCF     →  newCF
newC      →  newC
getC      →  getC
setC      →  setC
unpackCF  →  unpackCF
unpackC   →  unpackC
```

```
new    →  new
size   →  size
link   →  link
set    →  set
mkcur  →  mkcurrent
cur    →  getcurrent
get    →  get

ineg   →  negi
iadd   →  addi
imul   →  muli
isub   →  subi
idiv   →  divi
imod   →  modi
ieq    →  eqi
ilt    →  lti
igt    →  gti
ior    →  ori
iand   →  andi
ixor   →  xori
req    →  eqr
```

```
is-int           →  int?
is-frame         →  frame?
is-cf            →  cf?
is-continuation  →  continuation?
is-null          →  null?
is-code          →  code?

pop              →  rget?
print            →  print
printc           →  printc

tick             →  tick
tock             →  tock
forceGC          →  forceGC
debug-state      →  debug!
```

Figure 6.2: A mapping from Dynamix operations to Frame VM instructions

depending on the back end. This mapping for the Frame VM is given in Figure 6.2, with more details in Appendix B.

In addition to these back end specific primitives, there are also a couple of primitive operations that are provided by the Dynamix core. These operations are mostly related to operations on the scope graph of a program. Figure 6.3 informally describes their behavior. Later in this chapter this description is made concrete using examples and ultimately their formal semantics.

Operation	**Description**
`resolve`	Resolve a variable in a namespace using the scope graph to a path
`resolve-scope`	Resolve a variable in a namespace using the scope graph to a path to the declaring scope
`nop`	Do nothing. Returns the empty list of instructions
`length`	Get the length of a compile time list
`associate-index`	Associate a declaration with an index in its scope. By default this index is created automatically, but sometimes it is needed do enfoce a certain index.
`explode-string`	Convert a string to a list of character codes
`debug`	Prints the value of its argument to the console

Figure 6.3: Built-in Dynamix primitive operations

6.2 Compiling Tiger

The Tiger programming language was created by Appel [5]. It is a simple language with functions, let-bindings, arrays, records and various control flow constructs like loops.

In this section a partial implementation of the semantics of Tiger is discussed to show how Dynamix can be used. A full implementation of Tiger in Dynamix can be found in Appendix D. For the syntax and name binding rules, previous work by Vergu, Tolmach, and Visser [51] was used as a base.

6.2.1 Initial Evaluation Step

```
1  eval(Mod(exp)) =
2      v <- eval-exp(exp);
3      print(v);
4
5      callC(getC(curCF(), $ret), int(0))
```

Figure 6.4: The `eval` rule of the Tiger compiler

The entry-point of a Dynamix specification is the `eval` rule. This rule is applied to the AST of the program that is to be compiled. In the case of the used Tiger syntax, this makes that we are given a `Mod` element containing the top-level expression. The `eval` rule (Figure 6.4) therefore matches on this module element. In order to evaluate the expression, a second rule is used that is dedicated to evaluating expressions. The result of this evaluation is stored in the variable `v`.

This value is the final outcome of the execution of a Tiger program and has to be printed to the console. As this is quite a complex operation, the debug `print` primitive is used. This operation prints internal representations of the values (`IntV(42)` or `NullV()`), but is good enough for the purpose of this example.

The last step is to terminate the VM, which is done by returning the exitcode 0 from the initial control frame. For this the return continuation of the current control frame is gotten from the `$ret` continuation slot and invoked with as argument the integer value 0.

6.2.2 Simple Expressions

```
1  eval-exp(Int(v)) = return(int(v))
2
3  eval-exp(Plus(left, right)) =
4      v1 <- eval-exp(left);
5      v2 <- eval-exp(right);
6      return(iadd(v1, v2))
7
8  eval-exp(Var(name)) =
9      path = resolve(name, "Var");
10     return(get(cur(), path))
```

Figure 6.5: The `eval-exp` rules of some simple Tiger expressions

The first expressions that we add to the Tiger specification are simple expression like addition and integer constants. The Dynamix rules that these expressions add are shown in Figure 6.5.

Probably the simplest expression in Tiger is an integer constant. In the AST this is a node of the form `Int(`v`)`. Therefore the Dynamix rule that evaluates an integer constant has to

match this element. Using the primitive operation `int` the integer value from the AST can be converted to a integer on the Frame VM.

A more complex expression is the addition of two numbers (`Plus(`l`, `r`)` in the AST). The semantics of this operation can be written down in a big-step style. First the left and right expressions are evaluated to two values (`v1` and `v2`). Lastly these two values are added using a primitive integer addition operation (`iadd`). The result of the primitive addition is again returned as a result of the evaluation rule.

The third Tiger expression in Figure 6.5 is a variable reference `Var(`x`)` . As Dynamix is created with scope graphs in mind, getting the value for that variable starts with resolving the reference to its corresponding declaration. For this the primitive `resolve` is used. This primitive takes an occurrence and a namespace (in our case the name of the variable and the namespace `Var`) and produces a path in the scope graph. As this path is not a value that results from a computation on the VM but a value that becomes part of the produced bytecode, the value is a compile-time value. In Dynamix an explicit distinction is made between these two types of values. As an effect there are two binding operators. The "normal" binding for runtime values is the one used up to this point and the binding operator for compile-time values can be seen on line 9.

With scopes-as-frames in mind, we know that the value of the variable can be found by traversing the path found in the lookup from the current data frame (`cur`). The resulting value is then again returned from the rule.

6.2.3 Let Bindings

In the section on simple expressions variable references were added, but without declarations these are not very useful. In Tiger variable declarations are done in a let-binding. This let-binding is represented in the AST as `Let(binds, body)` , where `binds` is a list of binding constructs and `body` a list of expressions.

The evaluation rules for let-bindings are by far the most complex Dynamix rules we've seen so far. As shown in Figure 6.6, multiple rules are used that recursively call each other. Starting at the first rule, it can be seen that the evaluation of a let binding contains three components. On line 3 all the bindings are evaluated, on line 4 the body is evaluated to a value `res` and on line 5 the old/outer scope is restored.

The evaluation of the first two components is what make the rules for a let-binding interesting. Starting with evaluation of the body (line 9-15), we see two rule instances. These are used for two distinct cases in evaluating the list of expressions in the body of a let-binding. In the case where there is only one expression in the list, he first rule instance is selected. This instance evaluates the expression and the result is returned. In the case where we have multiple expressions, they all have to be evaluated and the result of the last expression has to be returned. For this reason the second rule instance evaluates the head of the list to a value and recursively calls itself with the tail of the list.

The evaluation rules for the bindings (line 18-30) follow a similar recursive procedure. The difference is that we now have the empty list of bindings as a base case and that we have two types of bindings that are distinguished. In the case of the empty list of bindings there is nothing that needs to be done. Therefore the body of this first instance of the `eval-let-binds` rule has the `nop` primitive as its body. This primitive does exactly nothing.

The second instance of the `eval-let-binds` rule does something we have not yet encountered. It constructs a new AST element for the variable declaration without a type (`VarDecNoType`) from one with a type (`VarDec`). Using this technique, the semantics of the two equivalent operations can be expressed using a single rule instance. This same technique can also be used to express, for example, the lazy binary `and` in terms of an if-expression by constructing the AST of the if and evaluating this new expression.

```
eval-exp(Let(binds, body)) =
    old_s <- cur();
    eval-let-binds(binds);
    res <- eval-let-body(body);
    mkcur(old_s);
    return(res)

// Body
eval-let-body([exp]) =
    v1 <- eval-exp(exp);
    return(v1)

eval-let-body([exp | tail]) =
    v1 <- eval-exp(exp);
    return(eval-let-body(tail))

// Bindings
eval-let-binds([]) =
    nop()

eval-let-binds([VarDec(name, _, val) | tail]) =
    eval-let-binds([VarDecNoType(name, val) | tail])

eval-let-binds([VarDecNoType(name, val) | tail]) =
    v1 <- eval-exp(val);
    scope <- new(int(1));
    link(scope, cur(), &P);
    mkcur(scope);
    set(cur(), resolve(name, "Var"), value);
    eval-let-binds(tail)
```

Figure 6.6: The `eval-exp` rules of let bindings in Tiger

The third rule instance is the rule instance that performs the actual binding. This rule first evaluates the expression to a value that must be bound to the given variable. After that, a new scope (data frame) is created of size 1 to store the value in. This new scope/frame is linked to the current scope using a parent edge P (`&P`) and set to be the current scope using `mkcur` . Lastly the value of the expression is stored using a similar method as getting the value in the case of a variable reference, but using a `set` instead of a `get` .

6.2.4 Conditionals

One of the major features of Dynamix is that it allows the programmer to describe complex control flow. In all the rules encountered up until now, the flow within a rule was always linear. But in the case of conditionals and loops, this flow needs to be way more complex.

For this purpose Dynamix uses quoted blocks and the rule continuations. A quoted block (with syntax inspired by MetaML) can be seen as a standalone piece of the evaluation with its own variables. In the case of the if rule instance (Figure 6.7), each of the branches gets their own execution block. The block `then_b` is for the valuation of the then branch and the block `else_b` for the else branch. As these blocks are no runtime values, the compile-time binding operator has to be used.

```
1  eval-exp(If(cond, then, else)); k(v1) =
2      !v1;
3
4      c <- eval-exp(cond);
5      jumpz(c, else_b, then_b);
6
7      then_b = <
8        ~v1 <- eval-exp(~then);
9        jump(~k)
10     >;
11     else_b = <
12       ~v1 <- eval-exp(~else);
13       jump(~k)
14     >
```

Figure 6.7: The `eval-exp` rules of an if expression

Because quoted blocks are completely standalone, they do not have direct access to values bound outside of the blocks. To get access to these values, values can be spliced in using the anti-quotation operator `~`. The same holds for the variables one assigns to in a quoted block. These values only exist within the lifespan of a block. The variable `v1` inside the block is therefore not the same variable as `v1` outside of the block (quoted blocks are *not* lexically scoped). When anti-quoting a variable that is assigned to, it must also be explicitly be declared globally using the exclamation mark operator.

Joining the two branches together seems impossible at first, using what we already know about the Dynamix language. This is because we know that rule instances always evaluate top to bottom and that there are now two potential bottoms. What we would want to say is that the last instructions in both quoted blocks become the new last instructions in the rule. This is however not practical, as this would imply that the code generation forks and starts appending generated code to both branches. It is therefore needed to create a common join point that is automatically assigned to be the last evaluation step of the rule.

Dynamix provides the *rule continuation* (`k` in Figure 6.7) as this common join point. When specified in the signature of a rule instance, quoted blocks can use jumps to this rule continuation to join the execution paths. In the case of an if expression, we also need to return a value from the rule. Luckily the rule continuation accepts an expression that, when evaluated, produces the value that is returned by a rule instance. The value produced by the if is already stored in the variable `v1` by either of the branches, so the value in this variable can simply be returned.

6.2.5 Loops

A second example of non-linear control flow within a single Dynamix rule instance is a while loop (Figure 6.8). Like in the previous example, quoted blocks are used to describe the semantics. In broad strokes the rule instance for evaluation of a while loop is similar to the construction of a while loop in Figure 4.6. Like in this example the code contains three parts: The first initiates the loop, the second checks the loop condition and the third executes the loop body.

The loop body and condition check have to be executed in a separate control frame. This is because the loop body does not behave the same as the program region outside the loop with respect to control flow. The cause for this is that a break statement inside the loop does

```
eval-exp(While(cond, body)); k(null()) =
    w_cond = <
      c <- eval-exp(~cond);
      jumpz(c, ~w_end, ~w_body)
    >;
    w_body = <
      scope <- new(int(0));
      link(scope, [], &P);

      mkcur(scope);
      v1 <- eval-exp(~body);

      mkcur(get(cur(), [&P]));
      jump(~w_cond)
    >;
    w_end = <
      callC(getC(curCF(), $break))
    >;

    w_cf <- newCF(cur());
    setC(w_cf, $break, curC(k));
    callCF(w_cf, w_cond)

eval-exp(Break()) =
    callC(getC(curCF(), $break))
```

Figure 6.8: Evaluation rules for a while-loop in Tiger

not behave in the same way as such a break outside of the loop. By definition, this is when a new control frame should be introduced.

Therefore a control frame must be created with a new continuation path when initiating the execution of the loop. This new path (`$break`) is used as the continuation that breaks loop execution, whether by having a break statement or by the loop condition failing. Like when joining the two branches of an if-statement, the rule continuation `k` can be used as the target of the break.

Evaluation of the condition is equivalent to the evaluation of the condition in an if-statement. The sub-expression is evaluated to a value and this value is compared to 0 to decide which quoted block to execute. The quoted block for the body (`w_body`) executes the expression of the body in a freshly created scope of size 0. This new scope is linked to the current data frame (the frame reached by following the empty path from the current data frame) using a parent edge. After evaluation of the body expression the old scope is restored, as the condition is checked in that scope. Lastly execution jumps back to the condition check. When this condition check fails, the `$break` continuation is invoked to terminate the execution of the loop.

To terminate the loop, the rule continuation `k` has to be invoked. By design, the `eval-exp` rule needs to return a value equal to the value of the evaluated expression. In the case of a loop, this returned value must be a unit value, which is represented by a null pointer on the Frame VM. Returning this value is done by providing it as an argument to the rule continuation.

A second expression that is related to loops is the break. By the way in which execution of a loop is constructed, the evaluation rule of this expression is just a single line. This is because the control frame that executes the loop already stores a continuation that terminates loop execution when invoked. Therefore the only thing that the evaluation of a break should do is to look up this continuation and invoke it.

6.2.6 Functions

In Tiger functions are defined in let-bindings. Therefore a rule instance has to be added to the let-binding rules from Figure 6.6. This new rule instance is shown in Figure 6.9.

Conceptually compiling a function declaration is a two-step procedure. First the body of the function is compiled to some code. After that a reference to the location of the code is associated to the function. Calling a function then looks up this reference and starts executing code at that location.

Compiling the function body is, again, done using quoted blocks. In this quoted block (`fun_body`) the body of the function is first evaluated to some value. This value is the returned value of the function and is given as an argument to the `callC` operation that calls the return continuation `$ret`.

The variable to which the quoted block was bound now contains a reference to the code produced by compiling the quoted block. As an effect we can store this reference in the scope graph at the location of the function declaration on line 8.

Up until now we gave Dynamix free reign over how it handled paths in the scope graph. Slots in data frames would automatically be assigned to variables and Dynamix would make sure these locations remained consistent. When variables are only referenced by name, this process is completely fine. It makes that we do not need to bother about which values are stored where as we can just resolve the name to get its location.

But in the case of a function call, arguments are rarely passed by name. Rather the ar-

```
1  eval-let-binds([ProcDec(name, args, body) | tail]) =
2      associate-args(args, 0);
3      fun_body = <
4        v1 <-eval-exp(~body);
5        callC(getC(curCF(), $ret), v1)
6      >;
7
8      set(cur(), resolve(name, "Var"), fun_body);
9      eval-let-binds(tail)
10
11 eval-exp(Call(name, args)); k(pop()) =
12     df <- new(int(length(args)));
13     frame-store[eval-exp](args, df, 0);
14     link(df, get(cur(), resolve-scope(name, "Var")), &P);
15     block <- get(cur(), resolve(name, "Var"));
16
17     cf <- newCF(df);
18     setC(cf, $ret, curC(k));
19     callCF(cf, block)
```

Figure 6.9: The semantics of functions in Tiger

```
associate-args([], _) = nop()

associate-args([FArg(name, _) | tail], idx) =
    associate-index(idx, name, "Var");
    associate-args(tail, idx + 1)

frame-store[x]([], _, _) = nop()
frame-store[x]( [exp | tail], frame, idx) =
    val <- x(exp);
    set(frame, [idx], val);
    frame-store[x](tail, frame, idx + 1)
```

Figure 6.10: Dynamix helper functions for function calls

guments are passed by index and the compilers make sure that the first provided value is mapped to the name of the first argument in the function declaration. Therefore it is needed to specify this mapping between indices and names.

In order to do this each argument of the function is associated with an index using the `associate-index` primitive operation. For this a helper function is needed that is shown in Figure 6.10. This rule recursively walks over the list of arguments starting at the given index (0 in this case) until the list is empty. In each step the argument at the head of the list is associated with the current index. Then the function is called recursively on the tail of the list with an index that is increased by 1.

Calling a function first creates a control frame for the function to execute in. This control frame has a new scope that has the current scope as its parent. In this new scope the argument values are stored by index. The procedure for storing the arguments uses the helper function `frame-store`. This function is parameterized by the expression evaluation rule `eval-exp`. As an effect, this rule can use the evaluation rule to evaluate each argument to a value before storing it[1]. The `frame-store` rule has a similar structure to the recursive function for associating the function arguments to indices. But instead of associating indices to names, it stores values at those indices.

6.3 Library Functions

Writing a Dynamix specification of a full language can become quite cumbersome, despite it abstracting away a lot of the boilerplate code needed for code generation and gluing the semantic rules together. This is because a lot of the primitive operations in Dynamix are very low-level (partially caused by the Frame VM being low-level).

It would however be nice to write more high-level semantic specifications. With these higher-level operations, it is not needed for the creator of the semantics to be familiar with the exact low-level details to be able to define the semantics of a language [9].

Furthermore, a lot of rules will be shared between different languages, as similar languages will implement the same language constructs in a similar way. It would therefore be useless to keep rewriting these rules again and again for each language. This is not only likely to introduce bugs, it also takes a lot of unnecessary time.

[1]Strictly speaking the rule being parameterized is not required, as the name of the evaluation rule is already known. It is however a very useful feature for writing generic rules. Therefore it is beneficial to use this feature in at least one of the examples.

Luckily Dynamix provides a mechanism which allows creating abstractions. Rules can be written that abstract over the primitives to produce more complex language concepts. One could for example write a rule for performing a generic function call with a list of arguments, or one for creating a new scope. Grouping the rules for these commonly used language constructs in a library would then make it possible to share the same rules between many language implementations.

Over time, when more language patterns are added to this library, the library will contain a lot of puzzle pieces from which the semantics of a language can be constructed. Many of these library rules can be seen as modeling fundamental constructs of programming languages. A lot of work has already been done in compiling a library of such fundamental constructs (funcons) [9, 10]. This library of funcons contains over 100 constructs found in various case studies of programming languages.

These funcons behave slightly different from rules in Dynamix. For example, a Dynamix rule is able to return a computation that does not result in a value. Funcons, in contrast, are defined in a way where such a computation produces a unit value [9]. As a result, both an if-statement that does not produce a value and the if-expression that does result in a value can be represented using a single funcon. In Dynamix one would need to create a rule for each case.

While there are differences between funcons and Dynamix rules, the funcon library would provide a good source of inspiration when selecting which constructs should be abstracted. Furthermore, it would be a good exercise to investigate the relation between Dynamix rules and funcons. Expressing the semantics of funcons in Dynamix would result in a compiler from a language defined in terms of funcons to the Frame VM. In fact Churchill et al. explicitly mention that specifying funcons in terms of a different modular semantics framework should be possible [10]. Despite this being an interesting research direction, no efforts have been put into this investigation due to time limitations.

6.4 Workflow

Having a good workflow for creating a set of semantic rules that describe a language is important. Especially when these semantics are executable, this workflow includes steps that are not directly related to the semantic description itself. In the case of Dynamix, the workflow starts at the parsing of a program and analysis of its AST. The AST and scope graph resulting from these two steps are the inputs for the code generation step that is defined by the semantics written in Dynamix.

Executable semantics frameworks often provide at least a part of these steps inside the framework. While this has the benefit that the steps are tightly integrated, it also means that significant effort needs to be put into refining these tools. Otherwise, the resulting workflow has a poor experience in the preliminary steps compared to the core of the framework [9]. By integrating Dynamix in the Spoofax language workbench, its implementation benefits from all the tools that are already available. Writing the parser of the language is handled by SDF3 and the analysis is performed by Nabl2 or Statix[2]. Both SDF3 and Statix are actively developed in other research projects, and all advancements made in those directly improve the full workflow.

The next step is the creation of the semantic rules themselves. This step is also aided by the integration in the Spoofax language workbench, as it provides useful editor features. These features include colorful syntax highlighting and buttons for execution of programs using the semantic rules. In the future, Dynamix could be extended with a strong type system that could provide error message and hints to the programmer.

[2]See `http://www.metaborg.org`

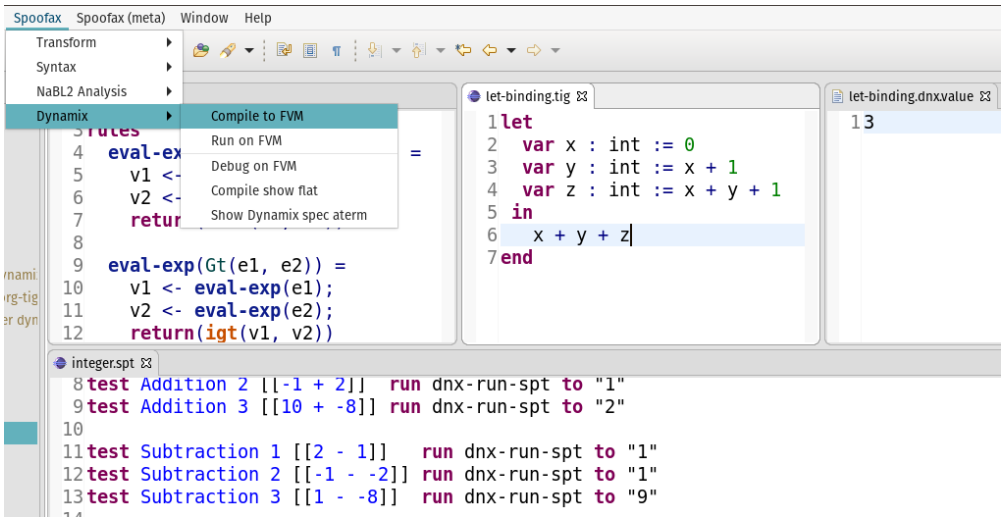


Figure 6.11: The workflow of developing a Dynamix specification in the Spoofax language workbench. The top row shows, from left to right, the windows where the Dynamix specification is written, an example program and the result of running the example program. The bottom row shows a number of SPT tests for verifying the description of the semantics.

When writing the semantics, it is also useful to be able to test the rules that are currently created. For this either manual testing can be used or automatic tests can be created. Manual testing follows the same steps as those an end-user of the language implementation will take. First a program in the source language is opened in Eclipse with the Spoofax and Dynamix plugins installed. A Dynamix menu-button is now added to the top bar of the editor which, when clicked, shows a button to run the currently open program. Clicking this button will compile the program using the semantic rules written in Dynamix and produce Frame VM bytecode. This code can then be executed to check the outcomes of a program.

A more systematic approach to testing the correctness of the semantics is to write a test suite that compiles and executes various test programs. The Spoofax language workbench has a built-in testing framework called SPT for which Dynamix provides a test strategy that performs the compilation and execution.

6.5 Semantics

After introducing the syntax of Dynamix and providing some example programs, an intuition about the working of the language should start to develop. One road to developing this intuition into a deep understanding of the language is to create/study more complex specifications written in Dynamix. Examples of these more complex specifications can be found all throughout the remainder of this report (chapter 7 and Appendix C, D and E). This section provides a second path to gain this understanding, by describing the formal semantics of the language.

6.5.1 Deriving the Core Language

For the definition of the semantics of Roger a few steps were provided that reduced the complete syntax to a more concise core language (subsection 4.4.2). This allowed for simpler semantic rules, while maintaining support for all language features. In this section, steps are defined to derive a similar core language for Dynamix.

The first step in a transformation to the core language has to do with the rule continuation. This continuation is optional and is used to create more complex control flow in a rule using

```
eval-exp(Plus(left, right)) =
    v1 <- eval-exp(left);
    v2 <- eval-exp(right);
    return(iadd(v1, v2))

eval(Mod(exp)) =
    print(eval-exp(exp));
    ret <- getC(curCF(), $ret);
    callC(ret, int(0))
```

⇒

```
eval-exp(Plus(left, right)); k(res) =
    v1 <- eval-exp(left);
    v2 <- eval-exp(right);
    res <- iadd(v1, v2);
    jump(k)

eval(Mod(exp)); k =
    print(eval-exp(exp));
    ret <- getC(curCF(), $ret);
    callC(ret, int(0))
```

Figure 6.12: Transformation of Dynamix rules to a version that always uses the optional rule continuation.

quoted blocks. But the fact that it is optional makes that semantic rules have to be created for both the case where it is used and the case where it is not. Luckily it is possible to remove one of these cases by making usage of the rule continuation required. This way only a single semantic rule is needed, and behavior is preserved[3].

Making rules adhere to the requirement that all rules must use the rule continuation is simple. For rules that did already make use of the rule continuation, nothing changes. The other rules can be split into two groups: The rules that return a value (the last instruction in the body is a call to a Dynamix-rule returning a value or a `return(..)`) and the rules that don't.

Both groups need a slightly different transformation (both are shown in Figure 6.12). In the case of the first group the returned value is added as argument to the rule continuation and the last instruction is replaced by a jump to the continuation. In the second group, it is sufficient to just add the jump to the rule continuation on the last line of the body. Adding this jump can be left out in this second case when the last instruction already transfers control to a different block.

The second step in the transformation to the core language separates the compile time and runtime bindings. As both types have distinct scoping rules (see Figure 6.13), this is a perfectly valid transformation. The only reason why a programmer might interleave both types is to improve readability by having declarations and references close together.

6.5.2 Semantic Rules

In this section the formal semantic rules of the Dynamix meta-compiler are shown and explained. This meta-compiler is given in its generic form, meaning it is not specialized to produce Frame VM bytecode. As an effect, only those primitive operations are shown that are present irregardless of the target language. Furthermore, the final output is not directly a valid AST for a Frame VM bytecode program but requires a post-processing step. In the implementation that targets the Frame VM, primitive operations are added and a post-processing step is used to produce a valid AST. These additional primitive operations are listed in Figure 6.2 and more detail about their semantics is located in Appendix B. In the post-processing step generic variable- and label names are replaced by names that are valid names in the Roger bytecode and the blocks are restructured to create a valid AST.

[3]The code generated by the Dynamix-derived compiler is different, but the effective behavior of the code is equivalent.

```
1  eval(exp); k(v1) =
2      !v1;
3
4      c <- eval(exp);
5      jump(block);
6
7      block = <
8        jump(~k)
9      >
```

⇒

```
eval(exp); k(v1) =
    !v1;
    block = <
      jump(~k)
    >;

    c <- eval(exp);
    jump(block)
```

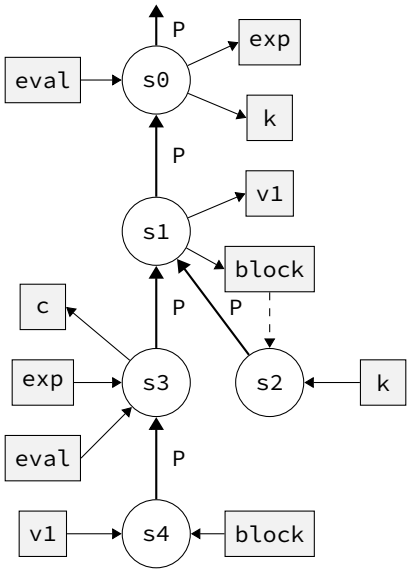


Figure 6.13: Separating the two different types of bindings. The scope graph is the same for both, justifying why this transformation is allowed

Conventions

The semantic rules for Dynamix use a style similar to the semantic rules of the Frame VM. All arrows are annotated by a letter in the fraktur font (e.g. $\mathfrak{m}$, $\mathfrak{p}$), marking the type of the rule that is applied. Capital letters are still used for constants provided in the initial state.

The set of all possible AST-elements or terms is called T. Possible terms $\tau \in T$ are a constructor with a name and a list of children $[\![d, u]\!]^\phi$, a tuple $(\tau+)^\phi$, a list $[\tau*]^\phi$, a number $z^\phi \in \mathbb{Z}$ or a string $s^\phi \in$ String. Here ϕ can be one of three things: $\phi = (ty, i)$ represents the type annotation and term index of the term, $\phi = \bot$ holds when this information does not exist. A variation of $\bot$, $\pm$, is used to indicate that this represents a runtime expression value which cannot be duplicated (as that would result in the expression being evaluated multiple times). When such a value is passed around, it must be made sure that it is first stored in a variable and that a reference to the variable becomes the value passed around.

The Dynamix specification that is used to compile the input AST with is stored in a map R. This map maps rule names to a list of rule instances r where each instance has the form $r = \langle q, a, k, b \rangle$ (rule generics parameters q, matchers a, rule continuation k and body b).

The names of primitive operations are stored in Π. This list is used to determine if an application of a rule needs to apply a primitive operation or if a matching rule in the Dynamix specification R needs to be found. Besides primitive operations that depend on the target language, there are also custom term constructors that differ for each target language. Strictly speaking, these custom constructors are not different from constructing terms the normal way, but they do provide a shorthand for commonly used constructors. In the case of the Frame VM these custom constructors are used for link- and continuation labels (`&P` and `$ret` respectively), instead of their longer forms `LinkLabel("P")` and `ContLabel("ret")`. To get this behavior, a mapping from the special characters used as prefixes to the longer internal names is used. In the semantic rules, this map is represented as C. Together, Π, C and the evaluation rules for the operations in Π define the behavior of Dynamix with respect to the target language of the meta-compiler.

The last capital letter that is used in the semantic rules is Γ. Where all previous capitals were constants provided in the initial rule, Γ does update during evaluation and is initial-

Used identifiers	Γ	$=$	$[x]$
Dynamix rules	R	$=$	$x \to [\langle q, a, k, b \rangle]$
Primitive operation names	Π	$=$	$[d]$
Custom term constructors	C	$=$	$g \to d$
Scope graph analysis	A		
The set of all terms	T		

Terms	τ	$=$	$[\![d, u]\!]^{\phi} \mid (\tau+)^{\phi} \mid [\tau*]^{\phi} \mid z^{\phi}, z \in \mathbb{Z} \mid s^{\phi}, s \in \text{String}$
Term annotations	ϕ	$=$	$(ty, i), i \in \mathbb{N} \mid \bot \mid \pm$
Bindings in a rule instance	σ	$=$	$x \to (\tau \mid x)$
Current block	χ	$=$	$[\![x, [\tau], \eta]\!]$
Set of created code blocks	β	$=$	$\xi \to \chi$
Expression value	η	$=$	$\tau \mid x \mid \bot$

Figure 6.14: The notation from the Dynamix syntax definition (Figure 6.1) is the base for the different letters used in the semantic rules of Dynamix. The letters in this figure are added to this base to provide ways in which internal data or the initial state is described.

ized in the initial rule to be the empty set. This set stores all identifiers that are created by the Dynamix meta-compiler and is used to make sure identifiers are never reused.

Lowercase Greek letters are again used for internal values of the execution. Two of these are used for terms and are already discussed before (τ and ϕ). The letter σ is used as the store that maps variable names to values. A special case of this store is σ_c, a store that only contains bindings to compile time values.

The output of the full meta-compilation is a map from names/labels to code blocks β. Each code block is a list of instructions, where each instruction can contain an expression. In the Dynamix meta-compiler these blocks are created one by one, where the block that is currently created uses the letter χ. When the start of a new block is found, this current block χ is flushed to β and the contents of χ are reset. In the end this creates a map of code blocks, each block containing a list of AST elements that represent the compiled instructions.

From this map the final output can be generated. This process depends on the target language, but likely contains two steps: First all variable names used in the generated code must be mapped to actual variable names. This process could be performed in a naive way, resulting in code which uses a lot of distinct temporary variables. Producing more efficient code can be done, but can be very tricky as optimal register allocation is NP-Complete [13].

$$N(\chi) = x \quad \text{where} \quad \chi = [\![x, \tau, \eta]\!] \qquad \text{(Current block name)}$$
$$C(\chi) = \tau \quad \text{where} \quad \chi = [\![x, \tau, \eta]\!] \qquad \text{(Current block code)}$$
$$V(\chi) = \eta \quad \text{where} \quad \chi = [\![x, \tau, \eta]\!] \qquad \text{(Current block value)}$$

$$V_{update}(\chi_1, \eta) = \chi_2 \quad \text{where} \quad \chi_2 = [\![N(\chi_1), C(\chi_1), \eta]\!]$$
$$C_{update}(\chi_1, \tau) = \chi_2 \quad \text{where} \quad \chi_2 = [\![N(\chi_1), C(\chi_1); \tau, V(\chi_1)]\!]$$

Figure 6.15: Projection and update functions for the currently constructing block

The current Frame VM backend uses such an naive approach and therefore produces far from optimal register naming, resulting in usage of hundreds of different registers instead of tens. A second step is to transform the generated list of blocks and lists of AST elements into an AST that is valid in the target language. The resulting AST from this can then either be directly executed, or pretty-printed to generate the source code.

Like was done in the semantics of Roger, the semantic rules of Dynamix also make use of projection and update functions. These functions are shown in Figure 6.15 and are used to interface with the currently constructing block χ. They allow to get the name, current list of instructions and expression value η from this block respectively for the projection functions. The update functions can be used to set the expression value and to add newly compiled instructions to the list of instructions in the block.

Initial Execution Steps

$$\boxed{R, \Pi, C \vdash \langle \tau, A \rangle \to \beta}$$

C-Program

$$\frac{\varnothing \vdash \langle R(\texttt{eval}), [\,], [\tau] \rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k, \Gamma_1 \rangle \qquad \Gamma_1, R, \Pi, C, A_1 \vdash \langle b, k, \sigma_c, \varnothing, [\![\texttt{MAIN}, [\,], \bot]\!] \rangle \xrightarrow{\mathfrak{b}} \langle \Gamma_2, A_2, \beta, \chi \rangle \qquad V(\chi) = \bot}{R, \Pi, C \vdash \langle \tau, A_1 \rangle \to \beta[N(\chi) \mapsto C(\chi)]}$$

Figure 6.16: Initial invocation of the meta-compiler. Given an AST τ, its binding analysis A and a Dynamix specification $\langle R, \Pi \rangle$, produces a list of named code blocks stored in a map β.

The initial execution step of compiling an abstract syntax tree (AST) using a Dynamix specification is conceptually very simple. Given an AST τ and its scope graph-based analysis A, a map containing all rules in the Dynamix specification R and a list of names of primitive operations Π, it produces a map β containing the generated code blocks (Figure 6.16). Generating these blocks is done in two steps. First the rule `eval`[4] is matched to the input AST, producing a list of argument bindings σ_c, the body b and rule continuation k of the matched rule. These bindings are then used to compile the body of the matched rule.

As a result of compiling the rule body we get a list of instructions, the unit value $\bot$ (as no value can be returned by the `eval` rule), an updated set of used names Γ and the created blocks β. This list of blocks is still missing one very important block: the last block we compiled in the recursive call (the block χ). Adding these instructions to β gives the final output of the compiler.

Rule Matching

When matching a rule, one rule needs to be selected from a list of rule instances. The instance that is selected is the first instance in the list of candidates for which *all* matchers in its signature produce a list of bindings. The 8 semantic rules in Figure 6.17 and 6.18 perform this matching by looping over all components of the rule signature.

Iterating over all rule instances is done in the rules `M-Match` and `M-Continue`. The first of these two rules tries to match the rule generics and the argument matchers with the given inputs. If these two matches succeed, a third step is to create a binding for the rule continuation using the $\xrightarrow{\mathfrak{r}}$-relation. All bindings created by these three steps are then presented as

[4]The `eval` rule is always the entry-point of a Dynamix specification.

$$\boxed{\Gamma \vdash \langle [r], q, [t] \rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k, \Gamma \rangle}$$

M-Match
$$\frac{\varnothing \vdash \langle q_1, q_2 \rangle \xrightarrow{\mathfrak{q}} \sigma_{c1} \qquad \sigma_{c1} \vdash \langle t, a \rangle \xrightarrow{\mathfrak{n}} \sigma_{c2} \qquad \Gamma_1 \vdash \langle k, \sigma_{c2} \rangle \xrightarrow{\mathfrak{r}} \langle \Gamma_2, \sigma_{c3} \rangle}{\Gamma_1 \vdash \langle [\langle q_2, a, k, b \rangle | r_t], q_1, t \rangle \xrightarrow{\mathfrak{m}} \langle \sigma_{c3}, b, k, \Gamma_2 \rangle}$$

M-Continue
$$\frac{\Gamma_1 \vdash \langle r_t, q, t \rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k, \Gamma_2 \rangle}{\Gamma_1 \vdash \langle [r|r_t], q, t \rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k, \Gamma_2 \rangle}$$

Figure 6.17: Semantic rules that select the first applicable Dynamix rule and create a set of bindings σ_c from rule arguments to values.

the result of the rule, together with the continuation and body of the matched rule instance. When any of the three steps fail, the `M-Continue` rule will be applied, which removes the first rule instance from the list of instances and recursively calls these two rules on the remainder of the list. In the case that the list of instances becomes empty, both rules will not be applicable and the selection of a rule instance fails.

Each of the three sub-steps consist of two cases, each described by a relation in Figure 6.18. The first pair is for creating bindings for the list of rule generics. This list contains variable names to which to bind the arguments to. As there are no additional requirements for creat-

$$\boxed{\sigma_c \vdash \langle [v], [v] \rangle \xrightarrow{\mathfrak{q}} \sigma_c}$$

Q-Empty
$$\frac{}{\sigma_c \vdash \langle [], [] \rangle \to \sigma_c}$$

Q-Fold
$$\frac{\sigma_c \vdash \langle t_1, t_2 \rangle \xrightarrow{\mathfrak{q}} \sigma_{c2}}{\sigma_c \vdash \langle [v_1|t_1], [v_2|t_2] \rangle \xrightarrow{\mathfrak{q}} \sigma_{c2}[v_2 \mapsto v_1]}$$

$$\boxed{\sigma_c \vdash \langle [\tau], [m] \rangle \xrightarrow{\mathfrak{n}} \sigma_c}$$

N-Empty
$$\frac{}{\sigma_c \vdash \langle [], [] \rangle \xrightarrow{\mathfrak{n}} \sigma_c}$$

N-Fold
$$\frac{\sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{\mathfrak{o}} \sigma_{c2} \qquad \sigma_{c2} \vdash \langle \tau_t, m_t \rangle \xrightarrow{\mathfrak{n}} \sigma_{c3}}{\sigma_{c1} \vdash \langle [\tau|\tau_t], [m|m_t] \rangle \xrightarrow{\mathfrak{n}} \sigma_{c3}}$$

$$\boxed{\Gamma \vdash \langle k, \sigma_c \rangle \xrightarrow{\mathfrak{r}} \langle \Gamma, \sigma_c \rangle}$$

R-KExp
$$\frac{x \notin \Gamma}{\Gamma \vdash \langle v(e), \sigma_c \rangle \xrightarrow{\mathfrak{r}} \langle \Gamma; x, \sigma_c[v \mapsto x] \rangle}$$

R-K
$$\frac{x \notin \Gamma}{\Gamma \vdash \langle v, \sigma_c \rangle \xrightarrow{\mathfrak{r}} \langle \Gamma; x, \sigma_c[v \mapsto x] \rangle}$$

Figure 6.18: Helper rules for matching rule arguments. The $\mathfrak{r}$ rules create the rule continuation binding, the $\mathfrak{q}$ rules match the rule generics and the $\mathfrak{n}$ rules match all arguments.

$$\boxed{\sigma_c \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_c}$$

O-Wildcard
$$\frac{}{\sigma_c \vdash \langle \tau, \_ \rangle \xrightarrow{o} \sigma_c}$$

O-Var
$$\frac{}{\sigma_c \vdash \langle \tau, x \rangle \xrightarrow{o} \sigma_c[x \mapsto \tau]}$$

O-Bind
$$\frac{\sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_{c2}}{\sigma_{c1} \vdash \langle \tau, x@m \rangle \xrightarrow{o} \sigma_{c2}[x \mapsto \tau]}$$

O-TermEmpty
$$\frac{d_1 == d_2}{\sigma_c \vdash \left\langle [\![d_1, [\,]]\!]^{\phi}, d_2([\,]) \right\rangle \xrightarrow{o} \sigma_c}$$

O-Term
$$\frac{d_1 == d_2 \qquad \sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c2} \vdash \left\langle [\![d_1, \tau_t]]\!]^{\phi}, d_2(m_t) \right\rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle [\![d_1, [\tau|\tau_t]]\!]^{\phi}, d_2([m|m_t]) \right\rangle \xrightarrow{o} \sigma_{c3}}$$

O-Type
$$\frac{\varnothing \vdash \langle ty_1, ty_2 \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c1} \vdash \left\langle [\![d, u]]\!]^{(ty_1, i)}, m \right\rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle [\![d, u]]\!]^{(ty_1, i)}, m : ty_2 \right\rangle \xrightarrow{o} \sigma_{c3}}$$

O-String
$$\frac{s_1 == s_2}{\sigma_c \vdash \left\langle s_1^{\phi}, s_2 \right\rangle \xrightarrow{o} \sigma_c}$$

O-Int
$$\frac{n_1 == n_2}{\sigma_c \vdash \left\langle n_1^{\phi}, n_2 \right\rangle \xrightarrow{o} \sigma_c}$$

Figure 6.19: The first set of matching rules for the rule arguments. Application of these rules could result in an updated set of bindings σ_c.

ing these bindings, the rules just fold over both lists creating a binding for the first elements until the lists are empty.

The second pair of rules is for matching the arguments to the matchers. Similar to the previous pair, the list of arguments is matched one-by-one until both the list of arguments and the list of matchers is empty. Each matching step matches an argument value η with its matcher. If matching succeeds a (potentially empty) list of bindings is produced.

The last two rules create a binding for the rule continuation. Execution of the rule body can use a jump to this continuation to join separate paths of execution back together. For this to work, a label must be created that will be used for the code block generated after applying the current rule. Therefore σ_c has to be extended with a mapping from the rule continuation variable name to the label. The two rules both handle one of the possible ways in which a rule continuation can be used. One for just the continuation, and one which also accepts an expression value.

Going one level deeper into the matching of a rule instance, we get the rules that perform the argument matching (Figure 6.19 and 6.20). Each of these rules describes a different type of matcher. The first rule (`O-Wildcard`) handles wildcard matches. Such a match will always succeed, no matter the provided input and will never introduce a new binding.

A variation of this rule is `O-Var`. This one always binds the provided value to a given variable. As a result the rule also never fails.

Sometimes it is needed to both match the structure of a given argument and be able to get

$$\boxed{\sigma_c \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_c}$$

O-TupleSingle
$$\frac{\sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_{c2}}{\sigma_{c1} \vdash \left\langle ([\tau])^{\phi}, ([m]) \right\rangle \xrightarrow{o} \sigma_{c2}}$$

O-Tuple
$$\frac{\tau_t \neq [\,] \qquad m_t \neq [\,] \qquad \sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c2} \vdash \left\langle (\tau_t)^{\phi}, (m_t) \right\rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle ([\tau|\tau_t])^{\phi}, ([m|m_t]) \right\rangle \xrightarrow{o} \sigma_{c3}}$$

O-ListEmpty
$$\frac{}{\sigma_c \vdash \left\langle [\,]^{\phi}, [\,] \right\rangle \xrightarrow{o} \sigma_c}$$

O-List
$$\frac{\sigma_{c1} \vdash \langle \tau, m \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c2} \vdash \langle \tau_t, m_t \rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle [\tau|\tau_t]^{\phi}, [m|m_t] \right\rangle \xrightarrow{o} \sigma_{c3}}$$

O-ListHeadSingle
$$\frac{\sigma_{c1} \vdash \langle \tau, m_1 \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c2} \vdash \langle \tau_t, m_2 \rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle [\tau|\tau_t]^{\phi}, [[m_1]|m_2] \right\rangle \xrightarrow{o} \sigma_{c3}}$$

O-ListHead
$$\frac{m_{1t} \neq \varnothing \qquad \sigma_{c1} \vdash \langle \tau, m_1 \rangle \xrightarrow{o} \sigma_{c2} \qquad \sigma_{c2} \vdash \langle \tau_t, [m_{1t}|m_2] \rangle \xrightarrow{o} \sigma_{c3}}{\sigma_{c1} \vdash \left\langle [\tau|\tau_t]^{\phi}, [[m_1|m_{1t}]|m_2] \right\rangle \xrightarrow{o} \sigma_{c3}}$$

Figure 6.20: The second set of matching rules for the rule arguments. This second set contains all rules for matching lists and tuples.

the full value. In this case a third variation of a variable binding matcher is used. Similar to the `O-Var` rule, this `O-Bind` rule produces a binding. But instead of never failing, this rule fails when the value bound to the variable does not match the matcher.

Matching the input with a term is defined in a recursive way. The first rule (`O-Term`) matches the name of the constructor and its first child. After that, a term is constructed that is equal to the initial term but with its first child removed. This new term is then recursively ran through the matching rules. At some point the list of children will become empty. In that case a second rule only has to verify that the constructor names of the terms are equal.

The next option to match on is on the type attached to a term using the `O-Type` rule. This type is part of the annotated AST that was fed to the Dynamix meta-compiler in the initial step as the variable A[5]. The attached type is also a term $\tau \in T$ and can therefore be matched with the desired type term using the matching rules described in this section. Any bindings that this might produce are ignored (though it might be interesting to try and think of cases where this information is actually needed in compilation). If matching the type was successful, the matching propagates to the term of which the type was checked.

The last two matching rules in Figure 6.19 are for matching strings and integer values. These two rules simply check that the provided value is equal to the expected value.

Matching a tuple or a list is a procedure that is similar to matching the children of a constructed term. In the case of a tuple the first rule (`O-TupleSingle`) matches a tuple with a

[5]In the current Dynamix implementation, this data is provided by the Nabl2-API ([32]).

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle b, k, \sigma_c, \beta, \chi\rangle \xrightarrow{\mathfrak{b}} \langle \Gamma, A, \beta, \chi\rangle}$$

B-Body
$$\frac{\Gamma_1 \vdash \langle b, \sigma_{c1}\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle \qquad \Gamma_2, R, \Pi, C, A_1 \vdash \langle b, k, \sigma_{c2}, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_2, \beta_2, \chi_2\rangle}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle b, k, \sigma_{c1}, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{b}} \langle \Gamma_3, A_2, \beta_2, \chi_2\rangle}$$

Figure 6.21: Semantic rules for compiling the body of a Dynamix rule in two steps: First the quoted blocks are pre-allocated, next the full body is compiled.

single element and propagates the matcher over this element. The case for multiple elements (`O-Tuple`) checks that there are indeed more than 1 elements in the tuple, matches the first element and propagates the checking of the tail. For a list, the procedure is exactly the same, except there is a case for the empty list (`O-ListEmpty`) instead of one for a single item. This is because an empty list can exist, but an empty tuple cannot.

The last two rules regarding matching are used for matching elements at the head of a list (`O-ListHead` and `O-ListHeadSingle`). First the head of the list is matched to the first matcher in the list of head matchers. This list is then reduced by one and a list head matcher is evaluated that is constructed using the tail of the argument list and the reduced list of matchers. When evaluation proceeds to the point where only a single head element is matched, the recursion stops and all elements remaining in the input list are bound to a variable.

Rule Body

After matching the rule instances to the given parameters, one instance is selected which will be evaluated. This evaluation consists of three steps: First, labels for quoted blocks and spliced register names have to be pre-allocated. This is because two quoted blocks may contain a jump to each other, creating a circular dependency. By pre-allocating the labels and registers, it is no longer a problem that values are still undefined when they are first used, allowing for these kinds of dependencies. The second step is to evaluate all instructions in the body of the rule. Lastly the rule continuation must be processed.

In the evaluation rules described in this section the second and third step are combined, resulting in two evaluation phases. The `D-Block` rule (Figure 6.5.2) shows exactly these phases. Given a body consisting of a list of instructions, the instructions are first run through

$$\boxed{\Gamma \vdash \langle [c], \sigma_c\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_c, \Gamma\rangle}$$

C-FoldEmpty
$$\frac{}{\Gamma \vdash \langle [\,], \sigma_c\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_c, \Gamma\rangle}$$

C-FoldQuote
$$\frac{x \notin Dom(\Gamma_1) \qquad \Gamma_1; x \vdash \langle c_t, \sigma_{c1}[v \mapsto x]\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}{\Gamma_1 \vdash \langle [v\texttt{= <}b\texttt{>}|c_t], \sigma_{c1}\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}$$

C-FoldDeclare
$$\frac{x \notin Dom(\Gamma_1) \qquad \Gamma_1; x \vdash \langle c_t, \sigma_{c1}[v \mapsto x]\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}{\Gamma_1 \vdash \langle [\texttt{!}v|c_t], \sigma_{c1}\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}$$

C-FoldOther
$$\frac{\Gamma_1 \vdash \langle c_t, \sigma_{c1}\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}{\Gamma_1 \vdash \langle [c|c_t], \sigma_{c1}\rangle \xrightarrow{\mathfrak{c}} \langle \sigma_{c2}, \Gamma_2\rangle}$$

Figure 6.22: Semantic rules for the quoted block pre-allocation step.

the pre-allocation steps ($\xrightarrow{\mathfrak{c}}$). After this step, the instructions are evaluated to a list of compiled instructions stored in the currently active block χ.

Figure 6.22 shows the three rules used in the pre-allocation step. The first, `C-FoldEmpty`, is the base-case where no more instructions are present and the pre-allocation is completed. The other three rules recursively fold over the list of instructions, reducing the amount of them in each step. In this process there are three cases (hence the three rules): Either the current instruction is a binding of a quoted block (`C-FoldQuote`), it is a variable declaration (`C-FoldDeclare`) or it is something different (`C-FoldOther`). In the last case there is nothing to pre-allocate, so we simply go to the next item in the list. In the other two cases an identifier has to be created that is bound to either the name of the quoted block or the name of the declared variable. This identifier is created by getting some value x, such that it is not in the set of currently used identifiers Γ. The identifier is then appended to Γ to make sure it will not be reused in later steps of the evaluation.

Statements

After the pre-allocation steps the actual evaluation of the body happens. The evaluation rules take a list of statements to evaluate and in each step the first instruction in that list is evaluated. The remainder is then processed in a next step until the list is empty. At this point the rule continuation is evaluated to produce the final output of the evaluation of the rule body. Evaluation rules for all the possible statements are given in Figure 6.5.2 and 6.5.2, those for the rule continuation are given in Figure 6.25.

The first statements that are found in the body of a rule instance are the compiletime bindings and variable declarations. The rule for a variable declaration (`D-CDeclare`) doesn't need to do anything and will simply pop the head of the list of statements. This is because all this instruction should do is to declare a variable and this is already done in the pre-allocation step.

The binding instruction does have to perform some actions when evaluating it. Again,

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle [c], k, \sigma, \beta, \chi\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma, A, \beta, \chi\rangle}$$

D-CBindQuote

$$\frac{\begin{array}{c}\sigma_2 = \{{\sim}x \mapsto \sigma_1(x) \forall x \in \sigma_1\} \\ \Gamma_1, R, \Pi, C, A_1 \vdash \langle b, \bot, \sigma_2, \beta_1, [\![\sigma_1(v), [\,], \bot]\!]\rangle \xrightarrow{\mathfrak{b}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ C(\chi_2) = [\,] \qquad V(\chi_2) = \bot \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle c_t, k, \sigma_1, \beta_2, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle\end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [v{=}\,{<}b{>}|c_t], k, \sigma_1, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}$$

D-CDeclare

$$\frac{\Gamma_2, R, \Pi, C, A_1 \vdash \langle c_t, k, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [!v|c_t], k, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle}$$

D-CBind

$$\frac{\begin{array}{c}\Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma_1, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{c}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ \chi_1 = \chi_2 \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle c_t, k, \sigma_1[v \mapsto \eta_1], \beta_2, \chi_2\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle\end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [v{=}e|c_t], k, \sigma_1, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}$$

Figure 6.23: Compiling the compile-time bindings of the Dynamix rule.

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle [c], k, \sigma, \beta, \chi\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma, A, \beta, \chi\rangle}$$

D-RBindExisting

$$\frac{\begin{array}{c} v \in \sigma \\ \Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle \qquad \eta = [\![\texttt{DNX-Assign}, [\sigma(v), V(\chi_2)]]\!]^{\perp} \\ V_{update}(\chi_2, \eta) = \chi_3 \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle c_t, k, \sigma, \beta_2, \chi_3\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_4\rangle \end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [v \leftarrow e | c_t], k, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_4\rangle}$$

D-RBindNew

$$\frac{\begin{array}{cc} v \notin \sigma & \Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ x \notin Dom(\Gamma_2) & \tau_1 = [\![\texttt{DNX-Ref}, [x]]\!]^{\perp} \qquad \tau_2 = [\![\texttt{DNX-Assign}, [\tau_1, V(\chi_2)]]\!]^{\perp} \\ C_{update}(\chi_2, \tau_2) = \chi_3 & \Gamma_2; x, R, \Pi, C, A_2 \vdash \langle c_t, k, \sigma[v \mapsto \tau_1], \beta_2, \chi_3\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_4\rangle \end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [v \leftarrow e | c_t], k, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_4\rangle}$$

D-Exp

$$\frac{\begin{array}{c} \Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma_1, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ V(\chi_2) = \perp \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle c_t, k, \sigma, \beta_2, \chi_2\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle \end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [e | c_t], k, \sigma_1, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}$$

Figure 6.24: Semantic rules for the runtime bindings and expression statements in a Dynamix rule.

quoted blocks have to be treated separately from all other expression that we might want to bind to a variable at compile time. Evaluating a quoted block creates a new code block from the body specified between the angle brackets. The process of evaluating this body to a block of code is very similar to how the body of a rule instance is compiled. Therefore the rule for evaluating a rule instance body ($\xrightarrow{\mathfrak{b}}$) is reused for evaluating the body of the quoted block. The difference is that evaluation of a quoted block does not continue code creation in the code block that is currently created (χ), but in a newly created block. A quoted block also lacks a rule continuation, therefore this value is set to $\perp$.

Before the body of a quoted block can be evaluated, a new store σ has to be created. Starting with an empty environment is easy, but this does not allow to splice in variables. To make spliced variables reachable requires us to add the currently visible variables to this new store prefixed by a tilde to indicate that they can be spliced in. This way any spliced variable (which start with a tilde) can now reach the value in the outer scope while the store seems empty for any non-spliced variable reference.

With this new store, the body of the quoted block can be evaluated. Resulting from this evaluation is an empty list of instructions and no expression value ($\perp$). This is because a code block generated from a quoted block is complete, i.e. the last compiled instruction from evaluating the quoted block is also the last instruction of the generated code block (no other instructions have to be added to complete the block). This generated block is already added to the set of code blocks in the last evaluation step, returning a new (empty) block which can be discarded.

Performing a normal compile time binding is a lot less complicated than binding a quoted block. The only thing that needs to be done is to evaluate the expression to some value η using the rules for expression evaluation ($\xrightarrow{\mathfrak{e}}$). We do however have to check that the compile time binding did not produce any runtime instructions by comparing the list of instructions

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle [c], k, \sigma, \beta, \chi \rangle \xrightarrow{\mathfrak{d}} \langle \Gamma, A, \beta, \chi \rangle}$$

D-KExp

$$\frac{V(\chi_1) = \bot \qquad \Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1[N(\chi_1) \mapsto C(\chi_1)], [\![\sigma(v), [\,], \bot]\!] \rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2 \rangle}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [\,], v(e), \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{d}} \langle \Gamma_2, A_2, \beta_2, \chi_2 \rangle}$$

D-K

$$\frac{\chi_2 = [\![\sigma(v), [\,], \bot]\!] \qquad V(\chi_1) = \bot}{\Gamma, R, \Pi, C, A \vdash \langle [\,], v, \sigma, \beta, \chi_1 \rangle \xrightarrow{\mathfrak{d}} \langle \Gamma, A, \beta[N(\chi_1) \mapsto C(\chi_1)], \chi_2 \rangle}$$

D-QuoteEnd

$$\frac{}{\Gamma, R, \Pi, C, A \vdash \langle [\,], \bot, \sigma, \beta, \chi \rangle \xrightarrow{\mathfrak{d}} \langle \Gamma, A, \beta[N(\chi) \mapsto C(\chi)], \bot \rangle}$$

Figure 6.25: Last step of rule body execution. There are three cases for the last step: Either there is a rule continuation with an expression, one without an expression or the body was part of a quoted block.

χ to the given input.

The second group of statements are the runtime statements. Semantic rules for these statements are given in Figure 6.5.2. There are only two valid statements in this group: Binding a runtime value to a variable and evaluating an expression.

For binding a value to a variable, two rules are needed. The first (`D-RBindExisting`) is for the case were we have already assigned a value to that variable before ($v \in \sigma$). Calling the expression evaluation rule with the given expression gives an updated current block χ. The expression value stored in this block can then be assigned to the variable location stored in σ. A generic `DNX-Assign` AST element is created to act as a variable assignment instruction. After the meta-compilation step, this AST element can then be replaced by an actual assignment in the target language.

Binding a value to a variable that has not been encountered before largely follows the same steps. The only exception is that we cannot use a store lookup to find the target variable to assign to. This variable has first to be created and added to the store. For this a name x is taken such that it is not yet in use ($x \notin \Gamma$) and a generic reference is created to this name and stored in the store.

The second valid runtime instruction is the evaluation of an expression (`D-Exp`). The call to evaluate the expression should not come as a surprise here and is fairly similar to its usages in the rules above it. One difference is that we require the output expression value to be $\bot$ (empty).

After all statements in the body are evaluated, the list of statements to evaluate has become empty. Now one of the three rules in Figure 6.25 is used to evaluate the rule continuation k. In the case where the statements were part of a quoted block instead of a rule instance, there is no rule continuation to apply. But as evaluation of any of these three rules mark the end of a code block, the instructions created up until this point have to be stored as a block in β. Creating this new block is done using the values stored in the current block χ and resetting this current block to an empty state.

When there is a rule continuation, it can be in one of two flavors: One with and one without an expression. In the latter case this expression has to be evaluated to a new current

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle e, \sigma, \beta, \chi\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma, A, \beta, \chi\rangle}$$

E-Var
$$\frac{V_{update}(\chi_1, \sigma(v)) = \chi_2}{\Gamma, R, \Pi, C, A \vdash \langle v, \sigma, \beta, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma, A, \beta, \chi_2\rangle}$$

E-Term
$$\frac{C \vdash \langle t, \sigma\rangle \xrightarrow{\mathfrak{t}} \tau \qquad V_{update}(\chi_1, \tau) = \chi_2}{\Gamma, R, \Pi, C, A \vdash \langle t, \sigma, \beta, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma, \beta, \chi_2\rangle}$$

Figure 6.26: Semantic rules for evaluation of variable references and term construction

expression value, in the first case this expression value is left empty.

Expressions

While describing the semantics of statements, the evaluation of expressions was used in many rules. These rules were not yet explained in detail. The only thing said about the evaluation rule for expressions was that the rules take an expression and produce a list of instructions and a value (that can be $\perp$). This section looks at how the three possible expressions (variable references, term constructions and calls) are evaluated.

As can be seen in Figure 6.26, evaluating a variable reference has a fairly simple rule. The variable name v is looked up in the store σ and the resulting value is returned. As no instructions are produced by a variable reference, the list of instructions χ given as an input does not change.

Evaluating an expression that constructs a term is deferred to a dedicated relation $\xrightarrow{\mathfrak{t}}$. This relation produces a term τ which is used in the result of evaluating the term construction rule `E-Term`.

The most complicated expression to evaluate is a call to an other rule. This is partially because there are three distinct types of calls, and partially because some extra work is required in evaluating the arguments to the call. In Figure 6.27 the three types of calls all have their own rule. The first rule, `E-CallPrim`, is used when the call is to a primitive operation. In this case the name of the called rule v is part of the list of primitive operations Π. As the primitive operations never accept generic arguments, it is checked that the given list is empty. Using the relation $\xrightarrow{\mathfrak{f}}$, the arguments to the function are then all evaluated to a list of values η_1. Together with the name of the called rule these values are passed to the rule that evaluates primitive operations. Resulting from this evaluation is an updated list of instructions and potentially a value. These new instructions are appended to the current list of instructions to get, together with the value, the result of the primitive call evaluation.

The next case of a call is a call to a rule somewhere else in the Dynamix specification. This is checked by verifying that the name of the called rule is in the active specification R. Furthermore it is checked that the name is not in the current store. This is done to ensure that generic rules (of which the binding is in the store) always take precedence over rules in the Dynamix specification. Next, the arguments are evaluated to a list of values, again using the relation $\xrightarrow{\mathfrak{f}}$.

The last part of the evaluation of this type of call is looking up the proper rule instance in R by matching the call arguments and evaluating the body of the matched instance. These two steps are similar to the behavior of the initial rule `C-Program`, with a number of differences. First, `eval` is no longer hard-coded to be the name of the called rule. Furthermore, the current code χ is already populated with a value and does not need to be initialized.

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle e, \sigma, \beta, \chi\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma, A, \beta, \chi\rangle}$$

E-CallPrim

$$\frac{v \in \Pi \qquad q = [] \qquad \Gamma_1, R, \Pi, C, A_1 \vdash \langle f, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{f}} \langle \eta_1, \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ A_2 \vdash \langle v, \eta_1\rangle \xrightarrow{\mathfrak{p}} \langle A_3, \tau, \eta_2\rangle \qquad C_{update}(\chi_2, \tau) = \chi_3 \qquad V_{update}(\chi_3, \eta_2) = \chi_4}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle v\, q(f), \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma, A_3, \beta, \chi_4\rangle}$$

E-Call

$$\frac{v \notin \Pi \qquad v \notin \sigma \qquad v \in R \qquad \Gamma_1, R, \Pi, C, A_1 \vdash \langle f, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{f}} \langle \eta_1, \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ \langle R(v), [(\sigma(x)\ \textbf{if}\ x \in \sigma\ \textbf{else}\ x), \forall x \in q], \eta_1\rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k\rangle \\ \Gamma_2, R, \Pi, C, A_2 \vdash \langle b, k, \sigma_c, \beta_2, \chi_2\rangle \xrightarrow{\mathfrak{b}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle v\, q(f), \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}$$

E-CallGeneric

$$\frac{v \notin \Pi \qquad v \in \sigma \qquad \Gamma_1, R, \Pi, C, A_1 \vdash \langle f, \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{f}} \langle \eta_1, \Gamma_2, A_2, \beta_2, \chi_2\rangle \\ \langle R(\sigma(v)), [(\sigma(x)\ \textbf{if}\ x \in \sigma\ \textbf{else}\ x), \forall x \in q], \eta_1\rangle \xrightarrow{\mathfrak{m}} \langle \sigma_c, b, k\rangle \\ \Gamma_2, R, \Pi, C, A_2 \vdash \langle b, k, \sigma_c, \beta_2, \chi_2\rangle \xrightarrow{\mathfrak{b}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle v\, q(f), \sigma, \beta_1, \chi_1\rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_3, A_3, \beta_3, \chi_3\rangle}$$

Figure 6.27: Evaluation rules for call expressions. Different cases are calling a primitive, calling a rule and calling a rule from the rule generics.

The biggest difference is that the initial call did not have rule generics which had to be dealt with. Similar to how there are two cases for calls, these generics arguments also have a case for when they originate from the rule generics and a case when they refer to rules from the Dynamix specification. In the first case, they are bound in the store and their actual value is looked up. In the second case, the identifier already points to a name in the Dynamix specification and no extra steps are required.

The last type of call is when the name of the called rule is bound in the store of the caller. The only difference between the rule for this case (`E-CallGeneric`) and the previous one is that we cannot directly lookup the name of the rule to match. Instead, this is done in two steps by first finding the actual name in the store. All other steps in the evaluation of the call are the same as in the previous rule.

Evaluating the arguments to a call is a bit more work than just evaluating each expression in the list to a value and returning a list of values. In reality, four different rules (Figure 6.28) are required to get the correct behavior. The last three rules in this figure take the the expressions from the list of arguments and evaluate them one by one.

Arguments to a call can evaluate to two types of values: terms and identifiers. The latter is handled by the `F-UnpackIdentifier` rule, which just returns the identifier. In the first case evaluation becomes a bit more complicated. This is because a value resulting from an expression might not really be a value. While it is fine in most cases to think about it as being a value, it is actually an AST element that computes a value when executed. As an effect this computation would be executed twice if it was passed directly to the called rule and this rule used the value twice. To combat this, it must be made sure that the values passed to the called rule are no complete computations, but rather a reference to a variable where the result of the computation was stored.

$$\boxed{\Gamma, R, \Pi, C, A \vdash \langle [e], \sigma, \beta, \chi \rangle \xrightarrow{\mathfrak{f}} \langle [\eta], \Gamma, A, \beta, \chi \rangle}$$

F-UnpackExpEmpty

$$\frac{}{\Gamma, R, \Pi, C, A \vdash \langle [\,], \sigma, \beta, \xi, \chi \rangle \xrightarrow{\mathfrak{f}} \langle [\,], \Gamma, A, \beta, \chi \rangle}$$

F-UnpackIdentifier

$$\frac{\begin{array}{c}\Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2 \rangle \\ V(\chi_2) = x \qquad x \in \Gamma_2 \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle e_t, \sigma, \beta_2, \chi_2 \rangle \xrightarrow{\mathfrak{f}} \langle \eta_t, \Gamma_3, A_3, \beta_3, \chi_3 \rangle\end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [e|e_t], \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{f}} \langle [x|\eta_t], \Gamma_3, A_3, \beta_3, \chi_3 \rangle}$$

F-UnpackTerm

$$\frac{\begin{array}{c}\Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, A_2, \beta_2, \chi_2 \rangle \\ V(\chi_2) = \tau^{\phi} \qquad \phi \neq \pm \qquad \Gamma_2, R, \Pi, C, A_2 \vdash \langle e_t, \sigma, \beta_2, \chi_2 \rangle \xrightarrow{\mathfrak{f}} \langle \eta_t, \Gamma_3, A_3, \beta_3, \chi_3 \rangle\end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [e|e_t], \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{f}} \left\langle [\tau^{\phi}|\eta_t], \Gamma_3, \beta_3, \chi_3 \right\rangle}$$

F-UnpackExp

$$\frac{\begin{array}{c}\Gamma_1, R, \Pi, C, A_1 \vdash \langle e, \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{e}} \langle \Gamma_2, \beta_2, \xi_2 \rangle \\ V(\chi_2) = \tau_1^{\pm} \qquad x \notin Dom(\Gamma_2) \qquad \tau_2 = [\![\texttt{DNX-Ref}, [x]]\!]^{\perp} \qquad \tau_3 = [\![\texttt{DNX-Assign}, [\tau_2, \tau_1^{\pm}]]\!]^{\perp} \\ C_{update}(\chi_2, \tau_3) = \chi_3 \qquad \Gamma_2; x, R, \Pi, C, A_2 \vdash \langle e_t, \sigma, \beta_2, \chi_3 \rangle \xrightarrow{\mathfrak{f}} \langle \eta_t, \Gamma_3, A_3, \beta_3, \chi_4 \rangle\end{array}}{\Gamma_1, R, \Pi, C, A_1 \vdash \langle [e|e_t], \sigma, \beta_1, \chi_1 \rangle \xrightarrow{\mathfrak{f}} \langle [\tau_2|\eta_t], \Gamma_3, A_3, \beta_3, \chi_4 \rangle}$$

Figure 6.28: Semantic rules for recursively evaluating rule arguments before calling the rule.

By the way terms are notated in the semantic rules, the term annotation ϕ can be used to distinguish when to store a computation in a variable and when not to perform this step. This is because $\phi = \pm$ when a term is an expression AST element[6]. The `F-UnpackExp` rule uses this property to only apply in the case of a computation value. First a new identifier is created and a generic reference term is created using this identifier. An assignment instruction is then created that uses the new reference as a target and the computation value as its value. This instruction is then added to the list of instructions in χ and the reference is added to the list of argument values.

Constructing Terms

A class of evaluation rules that was not really covered in the section on the evaluation of expressions was the evaluation of expressions that construct terms. The creation of the term was deferred to a dedicated relation, $\xrightarrow{\mathfrak{t}}$, that produces a term τ. In this section the internals of this relation are described by looking at its evaluation rules (Figure 6.29).

The first semantic rule is one for a variable reference (`T-Var`). This rule looks up the variable in the store and checks if the stored value is a valid term.

Constructing integer and string terms is done by the next two rules (`T-Int` and `T-String` respectively). These two rules take the input argument and annotate it with the empty an-

[6] A notable exception is a variable reference. This AST element is explicitly annotated with $\perp$ as labeling it as $\pm$ would result in unnecessary boxing and unboxing of a value.

$$\boxed{C \vdash \langle t, \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-Var
$$\frac{\sigma(v) = \tau \qquad \tau \in T}{C \vdash \langle v, \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-Int
$$\frac{}{C \vdash \langle z, \sigma \rangle \xrightarrow{\mathsf{t}} n^{\perp}}$$

T-String
$$\frac{}{C \vdash \langle s, \sigma \rangle \xrightarrow{\mathsf{t}} s^{\perp}}$$

T-CopyIndex
$$\frac{C \vdash \langle t, \sigma \rangle \xrightarrow{\mathsf{t}} \tau_1^{(ty_1, i_1)} \qquad \sigma(v) = \tau_2^{(ty_2, i_2)}}{\langle t\texttt{@}v, \sigma \rangle \xrightarrow{\mathsf{t}} \tau_1^{(ty_1, i_2)}}$$

T-CopyIndexBot
$$\frac{C \vdash \langle t, \sigma \rangle \xrightarrow{\mathsf{t}} \tau_1^{\perp} \qquad \sigma(v) = \tau_2^{(ty, i)}}{\langle t\texttt{@}v, \sigma \rangle \xrightarrow{\mathsf{t}} \tau_1^{(\perp, i)}}$$

T-Constructor
$$\frac{\tau = [\![d, [t,\ \forall x \in u \,|\, \langle x, \sigma \rangle \xrightarrow{\mathsf{t}} t]]\!]^{\perp}}{C \vdash \langle d(u), \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-Custom
$$\frac{\tau = [\![C(g), [v^{\perp}]]\!]^{\perp}}{C \vdash \langle g\ v, \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-List
$$\frac{\tau = [y,\ \forall x \in u \,|\, \langle x, \sigma \rangle \xrightarrow{\mathsf{t}} y]^{\perp}}{C \vdash \langle [u], \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-Tuple
$$\frac{\tau = ([y,\ \forall x \in u \,|\, \langle x, \sigma \rangle \xrightarrow{\mathsf{t}} y])^{\perp}}{C \vdash \langle (u), \sigma \rangle \xrightarrow{\mathsf{t}} \tau}$$

T-listHead
$$\frac{\tau_1 = ([y,\ \forall x \in u \,|\, \langle x, \sigma \rangle \xrightarrow{\mathsf{t}} y])^{\perp} \qquad C \vdash \langle t, \sigma \rangle \xrightarrow{\mathsf{t}} \tau_2}{C \vdash \langle [u \,|\, t], \sigma \rangle \xrightarrow{\mathsf{t}} \tau_1; \tau_2}$$

T-Add
$$\frac{C \vdash \langle t_1, \sigma \rangle \xrightarrow{\mathsf{t}} z_1 \qquad C \vdash \langle t_2, \sigma \rangle \xrightarrow{\mathsf{t}} z_2 \qquad z_1 \in \mathbb{Z} \qquad z_2 \in \mathbb{Z}}{C \vdash \langle t_1 \texttt{+} t_2, \sigma \rangle \xrightarrow{\mathsf{t}} (z_1 + z_2)^{\perp}}$$

T-Sub
$$\frac{C \vdash \langle t_1, \sigma \rangle \xrightarrow{\mathsf{t}} z_1 \qquad C \vdash \langle t_2, \sigma \rangle \xrightarrow{\mathsf{t}} z_2 \qquad z_1 \in \mathbb{Z} \qquad z_2 \in \mathbb{Z}}{C \vdash \langle t_1 \texttt{-} t_2, \sigma \rangle \xrightarrow{\mathsf{t}} (z_1 - z_2)^{\perp}}$$

Figure 6.29: Semantic rules for constructing terms $\tau \in T$

notation $\perp$ to make it a valid term. By definition s is a string and $z \in \mathbb{Z}$, so there is no need to verify this in the rules themselves.

A more interesting rule is `T-CopyIndex`. This rule constructs a term from argument t and then replaces its term index by the term index of a term in the store. It is however possible for the constructed term τ_1 to have no type annotations attached. In this case the term index is taken from the term stored in the store and the type information is left empty.

The next rule in Figure 6.29 is `T-Constructor`. This rule creates a term with a given name and children. These children are terms as well and are constructed using a list comprehension where each element in the list is evaluated to a term.

Besides the normal term constructor, there is the custom term constructor. This constructor is used as a shorthand for various components in the target language. In the case of Roger, these custom constructors are used for link labels and continuation labels. Therefore it is possible to create a link label `Link(P)` by writing `&P` in the Dynamix rules. Here & signals that it is a link label and therefore evaluation of the term constructs the AST element $[\![$ `LinkLabel, ["P"]` $]\!]^{\perp}$.

In the evaluation rule (`T-Custom`) exactly this behavior can be seen. The custom constructor type g is looked up in the map from custom constructor names to full names C and a term is constructed with the result of this lookup as its name. The name attached to the custom constructor as an argument is then converted to a string term by adding the empty

$$\boxed{A \vdash \langle v, [\eta] \rangle \xrightarrow{\mathfrak{p}} \langle A, \tau, \eta \rangle}$$

$$\text{P-Nop}\ \frac{}{A \vdash \langle \texttt{nop}, [\,] \rangle \xrightarrow{\mathfrak{p}} \langle A, [\,], \bot \rangle} \qquad \text{P-Length}\ \frac{n = |\,\eta^{\phi}\,|}{A \vdash \left\langle \texttt{length}, [\eta^{\phi}] \right\rangle \xrightarrow{\mathfrak{p}} \left\langle A, [\,], n^{\bot} \right\rangle}$$

$$\text{P-Explode}\ \frac{cs = [\texttt{UTF-16 value of}\, c, \forall c \in \eta]^{\bot} \qquad \eta \in \mathit{String}}{A \vdash \langle \texttt{explode-string}, [\eta] \rangle \xrightarrow{\mathfrak{p}} \langle A, [\,], cs \rangle}$$

Figure 6.30: Miscellaneous Dynamix primitive operations and their semantics.

annotation after which it is stored as the only child.

Besides constructors and the simple terms like strings and integers, there are two more types of terms. These two types of terms are created using the rules `T-List` and `T-Tuple`. Lists and tuples both contain a list of sub-terms that have to be created. Therefore they are created in a very similar way. In both rules a list comprehension is used to evaluate each element that they contain.

It is also possible to create a list by prepending a number of elements to an existing list. The rule that evaluates this case is `T-listHead`. This rule first evaluates all the items to prepend to a list of terms. After that, the list to prepend to is evaluated and both lists are concatenated to produce the final output.

The last two rules from Figure 6.29 are used to perform meta-arithmetic, arithmetic performed during compile time. This is useful when zipping a list with indices in a recursive way, where each step increments the index by 1. Without meta-arithmetic it would not be possible to code this increment into a Dynamix specification. In the semantic rules a left and a right argument are evaluated to integer terms. These two integers can than be added together or subtracted to perform the meta-arithmetic.

Primitive Operations

The only rules that are still missing in order to complete the semantics of Dynamix are the rules that define all primitive operations. In this section only the primitive operations of the core language of Dynamix are shown. This is done to both keep this section short and because the semantics of Dynamix are written down in such a way that they are independent of the target language. The primitive operations for Dynamix that target the Frame VM are shown in Appendix B.

Most of the primitive operations in the Dynamix core are related to performing scope graph queries. But there are three other primitives present (Figure 6.30). The first, `P-Nop`, does literally nothing. It does not produce any instructions and the returned value is the empty value $\bot$. While this sounds like a useless rule, it is actually quite useful. It is often used in the base-case of a recursive strategy.

The second primitive operation is `length`. This operation takes a list, tuple or string term as argument and returns an integer equal to its size. Applications of this primitive include getting the length of the list of bindings in a let-binding to calculate the amount of memory that is needed for storing the values.

$$\boxed{A \vdash \langle v, [\eta] \rangle \xrightarrow{\mathfrak{p}} \langle A, \tau, \eta \rangle}$$

P-ResolveSelf

$$\frac{\eta \in \text{String} \qquad \phi = (ty, i) \qquad A \vdash n_i^{D_\eta} \in \mathcal{D}(S) \qquad A \vdash p : S \longmapsto n_i^{D_\eta}}{A \vdash \left\langle \texttt{resolve}, [n^\phi, \eta] \right\rangle \rightarrow \left\langle A, [\,], [\![\texttt{DNX-Path}, [p]]\!]^\perp \right\rangle}$$

P-Resolve

$$\frac{\eta \in \text{String} \qquad \phi = (ty, i) \qquad A \vdash p : n_i^{R_\eta} \longmapsto n_j^{D_\eta}}{A \vdash \left\langle \texttt{resolve}, [n^\phi, \eta] \right\rangle \rightarrow \left\langle A, [\,], [\![\texttt{DNX-Path}, [p]]\!]^\perp \right\rangle}$$

P-ResolveScopeSelf

$$\frac{\eta \in \text{String} \qquad \phi = (ty, i) \qquad A \vdash n_i^{D_\eta} \in \mathcal{D}(S)}{A \vdash \left\langle \texttt{resolve-scope}, [n^\phi, \eta] \right\rangle \rightarrow \left\langle A, [\,], [\![\texttt{DNX-Path}, [[\,]]]\!]^\perp \right\rangle}$$

P-ResolveScope

$$\frac{\eta \in \text{String} \qquad \phi = (ty, i) \qquad A \vdash p_1 : n_i^{R_\eta} \longmapsto n_j^{D_\eta} \qquad A \vdash n_j^{D_\eta} \in \mathcal{D}(S) \qquad A \vdash p_2 : n_i^{R_\eta} \longmapsto S}{A \vdash \left\langle \texttt{resolve-scope}, [n^\phi, \eta] \right\rangle \rightarrow \left\langle A, [\,], [\![\texttt{DNX-Path}, [p_2]]\!]^\perp \right\rangle}$$

P-AssociateIndex

$$\frac{\eta \in \text{String} \qquad j \in \mathbb{N} \qquad \phi = (ty, i) \qquad A_1 \vdash n_i^{D_\eta} \in \mathcal{D}(S) \qquad A_2 = A_1 \textbf{ where } A_2 \vdash [j] : S \longmapsto n_i^{D_\eta}}{A_1 \vdash \left\langle \texttt{associate-index}, [n^\phi, \eta, j] \right\rangle \rightarrow \langle A_2, [\,], \perp \rangle}$$

Figure 6.31: Semantics of Dynamix primitive operations related to scope graph operations. These rules use slightly altered notation from the original scope graph paper [35]

The last primitive that is not related to scope graph queries is `explode-string`. While explosions are awesome, this function is quite ordinary. It just converts a string to a list of UTF-16 encoded characters.

The second group of primitive operations, shown in Figure 6.31, are all about working with a scope graph. The notation used for the scope graph operations is a slightly adapted form of the notation used in the original paper on scope graphs ([35]). A declaration of a term n at term index i in namespace η is written as $n_i^{D_\eta}$. Similarly a reference of a term n at term index i in namespace η is written as $n_i^{R_\eta}$. The set of declarations defined in a scope S is written as $\mathcal{D}(S)$. A last bit of notation is how paths in the scope graph are represented. A path p between a reference $n_i^{R_\eta}$ and a declaration $n_j^{D_\eta}$ is written as $p : n_i^{R_\eta} \longmapsto n_j^{D_\eta}$. This path is a list of link labels from scopes to scopes (the initial step from a reference to its scope is omitted). When the path is from a scope to a declaration in this same scope, the link label is replaced by a declaration index. This index is unique for each declaration in a scope, where the first declaration gets index 0 and the others get incrementing integer values.

The first operation in the category of primitive scope graph operations that is shown is `resolve`. Given an AST element that represents a reference in the scope graph and a namespace, this operation produces a path through the scope graph from this reference to its dec-

laration. One exception to this behavior is when the given AST element is not a reference to a declaration but the declaration itself. In this case the path from the scope of the declaration to the declaration is returned (i.e. the index of the declaration in the scope). Both of these cases have their own semantic rule. In the normal case `P-Resolve` applies and `P-ResolveSelf` is used in the second case.

Sometimes it is not needed to resolve a reference completely, and a path to the declaring scope suffices. In this case the primitive operation `resolve-scope` can be used. Again, this operation has two cases. When the occurrence that needs to be resolved is already a declaration (`P-ResolveScopeSelf`), the path to its declaring scope is empty. When the occurrence is a reference, the `P-ResolveScopeSelf` rule first resolves the complete path to the declaration. After this step the declaring scope is found and a second path is created from the reference to this scope.

The last primitive operation is `associate-index` . This operation is needed because the index of a declaration in a scope is not properly defined. Is the first declaration the declaration with the lowest term index? Or could it be the first AST element that was declared to be a declaration?

In many cases it is fine to let this ordering be undetermined, as long as it remains consistent. For example, it does not matter in which order the fields of an object are stored in memory, as long as every reference points to the correct field and the order of the fields never changes. A problem arises however when thinking about function declarations and calls. When a function is declared it defines some arguments. In the body of the function it still does not matter in which order these arguments are stored in memory as the correct location can be found by resolving the name in the scope graph. But when calling the function this does no longer work. This is because the first argument is not guaranteed to be at the first index and we have no way of finding out its actual location.

To fix this issue there needs to be a procedure that allows to specify the indices and the order of declarations. The `associate-index` is precisely this procedure. Given a declaration, its namespace, and a desired index, it makes sure that the given declaration will be at that index when resolving its path. In order to do this, a new analysis A_2 is constructed that is equal to the original analysis, except the location of the declaration gets its new location.

6.6 Implementation Differences

While the previous section described the behavior of Dynamix by giving its semantics, there are some differences between the semantics and the actual implementation. These differences do not matter for most of the behavior that a user will encounter, but it is important to write them down.

The first difference is what both versions consider to be a wildcard. While the semantics limit a wildcard to be a single underscore, the actual implementation sees all identifiers that start and end with an underscore as a wildcard matcher. As an effect `_name_` is a valid wildcard in practice, but not in theory. This behavior was left out of the syntax for simplicity and because it did not add any meaningful behavior.

A bigger difference is that the semantics describe a system that is very general and can be adapted to different target languages, where the actual implementation is more tightly coupled to the Frame VM. The decoupling was kept in mind during development of the language, but some design decisions made the implementation more tailored to work with the Frame VM. When writing down the formal semantics of the language, small architectural changes were made that made it easier to decouple the target language components.

An example of this coupling in the actual implementation is that the three compilation phases described in the semantics are actually just two phases in the implementation. Instead of producing a list of labeled code blocks which contain generic variable names, assignments

and references, the output of the first phase already contains valid Frame VM names. Furthermore the semantics have a very generic way in which custom term constructors are handled. In the actual implementation the two special constructors used by the Frame VM are hardcoded in the syntax and throughout the code generation.

While these differences do not show in most implementations of Dynamix specifications targeting the Frame VM, it will be required to refactor the implementation to a certain extend when a new target language is introduced. The semantics described in this chapter should be used as a guide in this refactoring.

Chapter 7

Scheme and Tiger in Dynamix

For the evaluation of Dynamix and the Frame VM two Dynamix specifications were written. The first is a Dynamix specification of a small subset of Scheme. This language subset was chosen to show off the ability of the Frame VM to execute call/cc.

The second Dynamix specification was written for Tiger. In chapter 6 a simplified version of this specification was shown as an example of how semantics of languages are described in Dynamix. The complete implementation of Tiger is used in this chapter to run a suite of Tiger benchmark programs. Measuring the performance of the VM is only a secondary objective, however. Investigating the operational completeness and coverage of both the VM and the language description in Dynamix are more valuable for now.

Besides running the benchmark, a second case study was performed. The goal of this case study was to investigate the modularity of a language specification written in Dynamix. To this end an exception handling mechanism and generator functions were added to the language.

For both of the languages only small snippets of the Dynamix specification are shown in this chapter. The complete implementations are found in Appendix C and D for Scheme and Tiger respectively.

7.1 Scheme

Scheme is a programming language that was created by Guy Lewis Steele Jr. and Gerald Jay Sussman as a dialect of Lisp. The focus of the language was to remove language weaknesses that made it appear that certain features were needed [45]. This implies that, in practice, very few language constructs exist.

One of the distinguishing features of Scheme is that it supports `call/cc`. This operation is a very general control flow construct and is shortly touched upon in subsection 3.2.1. Implementing a language with such general continuation support as Scheme on the Frame VM is a good test case for its support for first-class continuations.

Scheme saw a lot of different implementations in its history. Not all of these were compatible with each other, giving rise to different Scheme dialects. Some notable examples are MIT-Scheme [25] and Racket [20] (formerly known as PLT-Scheme). To combat the divergence of the different implementations, several Language revision reports were created [1, 28, 45].

The implementation in Dynamix is closest to the semantics of MIT-Scheme. This means that multiple definitions of the same variable are allowed and `(set! .. ..)` returns the old value of the variable. Using these semantics allowed for the easiest construction of small example programs that used call/cc. One point where the semantics differ from MIT-Scheme is in the evaluation order. Officially, the evaluation order is undefined for MIT-Scheme [25],

```
(define x #f)
(+ 1
    (call-with-current-continuation
        (lambda (cont)
            (set! x cont)
            -1
        )
    )
)                    ; 0
(* (x 20) 2)         ; 21
```

Figure 7.1: Call/cc captures up to the end of the enclosing top level expression. Calling the exception does not continue evaluation of the expression that invoked the continuation.

but in practice MIT-Scheme uses right-to-left evaluation. For the Dynamix implementation the more intuitive left-to-right evaluation order was implemented.

7.1.1 Call/CC Intricacies

A keen reader might have spotted some issues with the behavior of `call/cc` in the example from Figure 3.4. One of these is that the continuation that is stored should not be just adding the number 2, but also the rest of the program. With this in mind, the continuation should be something like $(+\,2\,\bullet;\,(\texttt{add-2}\,40);\cdots)$. Clearly invoking this continuation would result in an infinite loop that will never terminate.

The fact that this does not happen can be explained by the fact that Scheme secretly gives a continuation that is delimited to the current line. This has the effect that the execution of the program can conceptually be seen as if it was executed in a read-evaluate-print loop (REPL)[1]. The current continuation is not to finish execution of the current line followed by the successive lines, but to finish the current expression and go to the last non-executed line. Conceptually this is similar to executing in a REPL, where the continuation would be to wait for the next input.

Furthermore, invoking a previously stored continuation will abort execution of the current expression. This is simply because finishing that partially evaluated expression was not part of the current continuation when it was captured. This is why the example from Figure 7.1 will not return 42, but just 21.

7.1.2 Scheme Semantics in Dynamix

The most important Dynamix rules in the Scheme specification are the rules needed for compiling call/cc. It is however needed to first understand how lambdas and function calls are encoded in Dynamix, as parts of these rules are used for describing call/cc.

When performing a normal function call many things are known in advance, resulting in highly specialized code. It is, for example, always known where the code of the function body resides or how to link the scope/data frame of its body to other data frames. In the case of closures, this information is no longer statically known. Therefore this information must be stored at runtime in a closure object. These objects contain information about execution of their body (a code pointer) and an environment to execute in (a data frame). As an effect, the evaluation rules of lambda expressions in Scheme must create such a closure object (Figure 7.2).

[1] As a matter of fact, MIT-Scheme does not even allow running a whole file. The way this is done is by piping a file to the executable line by line.

```
3  eval-exp(Lambda(args, body)) =
4    df <- new(int(length(args)));
5    link(df, [], &P);
6
7    b <- <
8      eval-lamb-body(~body)
9    >;
10   clos <- new(int(2));
11   set(clos, [0], df);
12   set(clos, [1], b);
13   return(clos)
14
15 eval-exp(FunApp([func | args])); k(pop()) =
16   !closure;
17   call_cont = <
18     arg <- eval-first(~args);
19     callC(~closure, arg)
20   >;
21   call_clos = <
22     cf <- newCF(get(~closure, [0]));
23     setC(cf, $ret, curC(~k));
24     df <- unpackCF(cf);
25     frame-store[eval-exp](~args, df, 0);
26     callCF(cf, get(~closure, [1]))
27   >;
28   closure <- eval-exp(func);
29   jumpz(is-frame(closure), call_cont, call_clos)
```

Figure 7.2: Creating a closure object and calling it in Dynamix.

Where in Figure 6.9 the data frame of the function was created in the call, this operation is now performed in the function declaration in the case of a closure. This newly created data frame is stored as the first value in the closure data frame. The second component of this closure is a code pointer to the body of the compiled function. In the case of a specification written in Dynamix, this is a reference to the quoted block that evaluates the function body.

Calling the function is mostly the same as calling a normal function. The difference is that we do not know all information at compile time and have to load this dynamically from the data frame given as the closure. This also means that the argument values must be stored directly in the data frame contained in the closure. For this an auxiliary rule is used called `frame-store`. This rule takes a data frame and a list of expressions, evaluates each of the expressions and stores their values in consecutive slots of the data frame. A definition of this rule is given in Figure 6.10.

In Scheme continuations are also callable. This means that the function value provided as the function is not guaranteed to be a closure. Therefore a check is added on line 29 that either treats the input as either a closure or as a continuation.

Calling a continuation is relatively simple. This is because continuations can be called directly using `callC`. The only thing that needs to be added is the provided argument to the continuation. For this a Dynamix rule is used that matches the provided list of arguments to a list containing one element. This element is then evaluated to a value and this value is

```
3  eval-exp(Callcc(func)); k(pop()) =
4    cc <- curC(k);
5    clos <- eval-exp(func);
6
7    // Call the closure
8    cf <- newCF(get(clos, [0]));
9    setC(cf, $ret, cc);
10   df <- unpackCF(cf);
11   set(df, [0], cc);
12   callCF(cf, get(clos, [1]))
```

Figure 7.3: Call/cc implemented using Dynamix.

returned. For brevity, the implementation of this rule is not shown.

Now we are ready to look at the implementation of call/cc in Dynamix (Figure 7.3). From the rule for performing a function call, we already know that a continuation can directly be called. Therefore constructing the continuations does not require any extra steps to make it a closure. In fact, capturing the current continuation only takes a single line. It is not different than capturing the continuation for a return address when performing a function call.

The remaining lines of the Dynamix rule for call/cc (Figure 7.3) are needed to call the enclosed lambda with the continuation as only argument. These lines have a lot in common with lines 22-26 from Figure 7.2, with the exception being that there is only a single argument of which the value is already known. Therefore it does not have to be evaluated first.

The last rule that is discussed in the Dynamix specification of Scheme is the rule for emulating a read-evaluate-print loop (REPL). As discussed earlier in this chapter, the continuation of a line in Scheme is delimited by the line ending. The Frame VM does currently not support delimited continuations (see chapter 9), but a workaround is to simulate a REPL. As an effect the continuation of a line is not to continue to the next line, but to go to the next line that was never evaluated.

To simulate this behavior, a few steps have to be taken. When execution of a line is started a variable is stored that points to the next line. As continuations never contain the start of a line, this makes that the variable will always point to the next line that was never executed. Thus, when execution of a line completes, this variable can be used to jump execution to the correct line.

The variable that stores this location must be reachable by all locations where it might be referenced from. Therefore, this variable is stored in the root scope of the scope graph in the location corresponding to the declaration `__NEXT__`. Because it is allocated in the scope graph, the location of the variable can be found easily using a call to `resolve`.

In Figure 7.4 one of the occurrences of this implementation is shown. Here the rule that

```
3  eval-top(exp); k =
4    line-f <- get(cur(), resolve-scope("__NEXT__"@exp, "Line"));
5    set(line-f, [0], k);
6    print(eval-exp(exp));
7    jump(get(line-f, [0]))
```

Figure 7.4: Dynamix implementation of Scheme REPL simulation.

evaluates a line no longer ends on an implicit jump to the rule continuation. Now execution of the code needs to jump to the code point stored as the next line to execute. Before evaluation of a new line, this variable is then updated with the new location.

7.1.3 Coverage/Correctness

As the Dynamix specification of Scheme mostly focused on implementing call/cc, the coverage of the specification is low. The features that are supported are: call/cc, lambdas, function calls, cons-lists, car/cdr, define, set!, if and simple binary expressions[2].

Verifying the compiler and execution of programs was done by writing numerous example programs and checking their solutions with different Scheme compilers. But as each different Scheme implementation is slightly different and because the Dynamix specification uses a slight variation of the MIT-Scheme semantics, verifying the outputs is more complex than just comparing results. As an effect it also relied heavily on my intuition of how things should be executed to select the correct result. Many aspects of these test programs were later transferred to unit tests. These tests were then used to (regression) test the implementation of the Scheme semantics.

While it would have been better, in terms of showing correctness, to pass a test suite for one of the Scheme compilers, this was not as simple as it sounds. First the Dynamix implementation should have been constructed to follow the semantics of MIT-Scheme precisely. But most importantly, these test suites are huge and test a lot of language features that are not supported by the compiler described in this case study. Furthermore, these tests were ran inside a testing harness written in Scheme. As an effect, adapting an existing test suit was more work than writing a new, smaller and simpler one.

7.2 Tiger Dynamix Specification

The Tiger programming language was created by Andrew Appel [5]. It is a simple language with functions, let-bindings, arrays, records and various control flow constructs like loops.

In this chapter a subset of the implementation of the semantics of Tiger is discussed. In chapter 6 the foundation of the full implementation was already shown as an example on how a Dynamix specification is written. For the syntax and name binding rules, previous work by Vergu et al.[51] was used as a base. Only small changes were required, making it easier to just focus on the implementation of the semantics. After implementing the semantics of Tiger, two extensions of the language are covered. These extensions add exception handling mechanisms and generator functions to the language.

7.2.1 Coverage/Correctness

The Tiger specification written in Dynamix is not complete, but covers all important language aspects. This means that the specification shows all the concepts, but might not cover all instances of these features. For example, the binary *or* expression is not implemented, but the binary expressions for integer addition and integer comparison are. Of the over ten built-in functions (e.g. `printi`, `concat`, `size`) only a few have an actual implementation in Dynamix.

The subset of supported features was decided by wanting to run Tiger example files and a benchmark, while also covering all interesting control-flow constructs. These example files are the files Andrew Appel provides as test cases for Tiger [6, 5]. The benchmark files are

[2] While these binary expressions can be overwritten in Scheme with other functions, the Dynamix implementation does not allow this.

	Dynamix/FVM	Hand
queens	4.55	0.0013
list	82.46	0.0187
towers	51.80	0.0128
sieve	29.47	0.0043
permute	65.22	0.0156

Figure 7.5: Execution time in seconds for Tiger benchmark programs. The columns show the Dynamix compiled code and a handwritten Tiger interpreter created by Vergu, Tolmach, and Visser [51].

discussed in a later section, but consist of Tiger translations of Java files from the *are-we-fast-yet* benchmark repository [30]. These translations were made for a paper that explored the scopes-as-frames paradigm for use in improving the performance of meta-interpreters [51].

In short this means that the Dynamix specification of Tiger supports let-bindings, function calls/declarations, records, arrays, binary expressions, unary expressions, some build-in functions and control constructs like if, while and for. As a potential future extension, Demaille and Levillain describe a version of tiger with classes [14]. This extension could be a good case study for describing the semantics of dynamic dispatch.

To show correctness of the implementation, the output of running all example programs was verified using two reference guides [5, 14] and the Tiger interpreter created by Vergu, Tolmach, and Visser [51]. In addition, tests were added to the Tiger project to test the behavior and have a means of regression testing while working on the specification.

7.2.2 Benchmarking Tiger

While performance was never a focus in this project, it is beneficial to run a Tiger benchmark using the Dynamix meta-compiler and the Frame VM. This would not only show how slow the execution speeds are (spoiler: very slow), but more importantly it would show the ability of the Frame VM to run real-world programs.

As a benchmark a test set from a paper by Vergu, Tolmach, and Visser [51] was used. This benchmark included a hand written interpreter for Tiger to which the performance of the Dynamix-spec is compared.

To measure the execution speed, three instructions were added to the Roger bytecode language. The first instruction, `forceGC`, invokes a garbage collection run. Measuring the execution times can then be done by starting and stopping a timer using `tick` and `tock` respectively. When the timer is stopped, the elapsed time since its start is printed to the console.

The expectation is that the execution speeds for the Frame VM will be fairly slow. An early indication of this was already discovered when compiling more complex Rust programs to Frame VM bytecode [11]. The root cause for this is the implementation of the Frame VM itself. The VM is currently implemented as an interpreter implemented in Stratego. Being an interpreter is already costly, but even more so when the interpreter is implemented in a known slow language. Furthermore, the code generated by a Dynamix compiler is poorly optimized. With simple optimizations, it should be possible to shorten the code by a significant amount. Some of these optimizations are discussed in the future work section (chapter 9).

A last factor that severely limits the performance of the VM is that continuations are relatively expensive. This cost scales with the amount of register values used, and the inefficient code generated by the Dynamix meta-compiler uses a lot of unneeded registers. As an effect all function calls become relatively expensive.

Therefore, we can take an educated guess on the performance compared to native speed. A guess of a couple orders of magnitude of lost execution speed, would not sound too shock-

```
let
  var x : int := 1
in
  try (
    if x = 0 then
      throw -1
    else
      printi(12)
  ) catch ex in (
    printi(ex)
  )
end
```

```
try (
  try (
    throw -1
  ) catch ex in (
    throw (ex + 1)
  )
) catch ex in (
  printi(ex)
)
```

```
for y := 1 to 5 do (
  try (
    if y > 2 then
      throw 42
  ) catch ex in (
    break
  )
)
```

Figure 7.6: Three code examples of try-catch expressions in Tiger. The example on the left is a simple case where, depending on the value of `x`, an exception will be thrown and printed to the console. The example in the middle shows re-throwing an exception after catching it. The last example shows the interaction between try-catch and a loop.

ing. Looking at the benchmark results as shown in Figure 7.5, it is clear that this educated guess was not all that wrong when comparing to the handwritten interpreter.

But as said before, measuring the performance was never the main goal of this investigation. The goal was to show that we are able to compile and run the benchmark programs using the Dynamix meta-compiler and the Frame VM. As the benchmark was able to run and produce valid results this goal can be considered as accomplished.

7.2.3 Extending Tiger Control Flow

One of the strengths of the Frame VM is the modularity of describing control flow. As an effect it becomes easy to extend an existing language specification with more control flow constructs without them interfering with the existing semantics of the language. To show this, the next sections add generator functions and an exception handling mechanism to Tiger. The existing specification, described in the previous section, does not require any changes to support these new features.

In this section, the focus is put on the changes to the Dynamix-specification of Tiger. Work on extending the syntax, name binding rules and type checking is not discussed.

Exceptions

The added language constructs for exception handling are a try-catch block and a throw statement. Syntax for these constructs is shown using some example programs in Figure 7.6. The exceptions in this language extension are simple integer values. It would be possible to introduce a proper exception type, but this only complicates the examples.

The newly added Dynamix rules for `TryCatch` and `Throw` are shown in Figure 7.7. Though the rules might seem complicated at first, they are relatively straight-forward adaptations of the bytecode instructions for a try-catch and throw described in subsection 4.3.6. Lines 13-17 allocate the data frames for the two branches and link them to the current scope. Allocating a control frame for each of the branches and setting the correct continuations for those control frames is done on line 19-27. The control frame for the try block gets a new exception handler (the handler control frame) and both get a `$next` continuation that joins execution of both branches.

```
eval-exp(TryCatch(try_exp, Var(var), catch_exp)); k(null()) =
    try_b = <
        v <- eval-exp(~try_exp);
        callC(getC(curCF(), $next))
    >;
    catch_b = <
        val <- pop();
        set(cur(), resolve(~var, "Var"), val);
        v <- eval-exp(~catch_exp);
        callC(getC(curCF(), $next))
    >;

    try_df   <- new(int(0));
    catch_df <- new(int(1));

    link(try_df  , cur(), &P);
    link(catch_df, cur(), &P);

    try_cf   <- newCF(try_df);
    catch_cf <- newCF(catch_df);

    next <- curC(k);

    setC(try_cf  , $ex  , newC(catch_cf, catch_b));
    setC(try_cf  , $next, next);

    setC(catch_cf, $next, next);

    callCF(try_cf, try_b)

eval-exp(Throw(exp)) =
    v <- eval-exp(exp);
    callC(getC(curCF(), $ex), v)
```

Figure 7.7: Dynamix rules for the `TryCatch` and `Throw` AST nodes.

```
eval-exp(Divide(e1, e2)); k(idiv(v1, v2)) =
    !v1;
    throw_b = <
        callC(getC(curCF(), $ex), ~v1)
    >;

    v1 <- eval-exp(e1);
    v2 <- eval-exp(e2);
    jumpz(ieq(v2, int(0)), k, throw_b)
```

Figure 7.8: A safe division operator that throws an exception on a division by zero.

```
let
  generator gen(n : int) : int = (
    yield n;
    yield (n + 1);
    yield (n + 4)
  )
  in

    for i in gen(2) do (
      printi(i)
    )


end
```

```
let
  generator gen() : int = (
    yield 42;
    throw 1

  )
  in
    try (
      for i in gen() do (
        printi(i)
      )
    ) catch ex in (
      printi(ex)
    )
end
```

Figure 7.9: Two code examples of generators in Tiger. The example on the left shows a generator function that yields all of its arguments in order. The rightmost example shows a generator working in conjunction with exceptions.

Besides adding the exception handling constructs, we could also change the semantics of existing language features to take advantage of exception handling. An example of this is a safe-division operator that will throw an exception on a division by zero instead of crashing the machine. A Dynamix rule modeling this behavior is shown in Figure 7.8. This rule specifies that the dividend and divisor are first evaluated to a value. Then the value of the divisor is compared to the integer 0. If they are equal we branch to the block `throw_b` which throws an exception with the dividend as value. otherwise, we branch to the rule continuation `k` with the result of the actual division.

Generator Functions

A second control structure that was added to the Tiger language are generator functions. The syntax and intended behavior of the generator functions is heavily inspired by Python generators [39]. Two examples of how generators can be used in the extended version of Tiger are shown in Figure 7.9. From these examples it becomes clear that three language constructs had to be added to the language. In addition to a new keyword for a generator declaration and a yield instruction, a new way of writing a loop was added. This for-each-like looping construct unwinds the provided generator and executes its body with each of the yielded values.

The simplest of the newly added rules is the rule for yield statements (see Figure 7.10). This rule looks similar to a return from a normal function. The difference lies in the fact that it does not just return a single value. The yield statement also returns its own continuation. Invoking this continuation will allow the caller to resume execution of the generator.

Yielding also does not call the normal return continuation `$ret`. This is because this continuation is already used for termination of the generator. Instead a new continuation `$yield` is used.

Constructing the generator when calling it for the first time is relatively straightforward. At its core it is the same as a normal function declaration (Figure 6.9). The main difference is that the body of the generator consists of two quoted blocks instead of the one quoted block used in the function rule. The first block is the generator initialization that returns a continuation that can be invoked to start the generator. The second part is similar to a normal

```
1  eval-let-binds([GenDec(name, args, _, body) | tail]) =
2    // ...
3    gen_init = <
4      callC(getC(curCF(), $ret), curC(~gen_body))
5    >;
6    gen_body = <
7      v1 <-eval-exp(~body);
8      callC(getC(curCF(), $ret))
9    >;
10   set(cur(), resolve(name, "Var"), gen_init);
11   eval-let-binds(tail)
12
13 eval-exp(Yield(exp)); k(null()) =
14   val <- eval-exp(exp);
15   callC(getC(curCF(), $yield), val, curC(k))
```

Figure 7.10: Dynamix rules for generator function declarations and yield statements

function body, as it just compiles the body. But as a generator does not return a value, the last line of the main body performs a void return.

The Dynamix rule for unwinding the generator in the for-in loop is a bit more complex than the other two rules for generators. The rule, as shown in Figure 7.11, consists of three main parts. A first part evaluates the generator expression to a continuation and creates a new control frame for the body of the loop (`for_cf`). Like the control frame of the body of a while loop (Figure 6.8) this control frame has a `$break` continuation (line 26).

Executing the body of the for-in loop first initializes the generator in such a way that its `$yield` handler executes the `exec_body` block. This block gets the two yielded values from the generator: a continuation to continue execution of the generator and the yielded value. This value is stored in the iteration variable of the loop using a `set` on line 12, after which the compiled body of the loop is executed. When the body finishes, the continuation of the generator is called again (line 14).

```
eval-exp(ForIn(Var(var), gen_exp, body)); k(null()) =
  var_path = resolve(var, "Var");
  exec_body_init = <
    gen <- get(cur(), ~var_path);
    gen_cf <- unpackC(gen);
    setC(gen_cf, $yield, curC(~exec_body));
    callC(gen)
  >;
  exec_body = <
    gen <- pop();
    val <- pop();
    set(cur(), resolve(~var, "Var"), val);
    v <- eval-exp(~body);
    callC(gen)
  >;
  gen <- eval-exp(gen_exp);
  for_df <- new(int(1));
  link(for_df, cur(), &P);

  exit_c <- curC(k);
  set(for_df, var_path, gen);

  setC(unpackC(gen), $ret, exit_c);

  for_cf <- newCF(for_df);
  setC(for_cf, $break, exit_c);
  callCF(for_cf, exec_body_init)
```

Figure 7.11: Dynamix rules for for-in loops

Chapter 8

Related work

The work described in this thesis consists of two components. In the first part the Frame VM is described. This VM uses the scopes-as-frames paradigm as a basis for its memory model and control frames for its runtime model for control flow. As an effect the Frame VM has support for first-class continuations, giving it support for a large number of control flow constructions. The Frame VM is however not the only system that sought after a runtime model that supported continuations. Other work in this area is described in section 8.1.

The second component of this thesis was the Dynamix language. This language was designed to allow programmers to write a semantic specification of a programming language. This specification would then be used to run a given program (in this case by compiling to Frame VM bytecode). Many other frameworks and tools have been developed over the years with similar goals. Some of which are described in section 8.2.

Lastly, work related to the meta-programming aspects of Dynamix is shown in section 8.3. This includes a classification of Dynamix in the spectrum of meta-programming systems.

8.1 Compiling With Continuations

The Roger bytecode language is not the only language that is designed with first-class continuations and continuation capturing mechanics. Most notably, Scheme and its dialects have put a lot of research in to developing compilation and runtime strategies for handling continuations.

Early Scheme compilers took a heap-based allocation strategy for both their call frames and their environment. This was done to simplify the creation of continuations and closures respectively. It did however severely limit the performance, by making variable references and procedure calls more expensive [16].

The approach of heap-allocating call frames and environments is similar to what the Frame VM does. Heap-allocated environments share properties with heap-allocated data frames. Heap-allocated call frames, and how these can encode continuations, are very similar to how control frames and continuations interact in the design of the Frame VM.

Despite the heap based Scheme implementations and the Frame VM being similar, current research in Scheme compilers has a different focus than the work described in this thesis. Where the work in this thesis is more focused on finding a good model, the Scheme community focused on performance and language features. Dybvig improved the performance by stack-allocation of call frames, with the expense of continuations being more computationally expensive [16]. The Chez-Scheme runtime resulting from this was then slightly changed to support the Racket language, improving the performance of this Scheme dialect [22]. Other researchers used JIT-compilation techniques to improve the performance of Racket [8].

Orthogonal to the work into improving the execution speed, research is performed into composable use of control. This is related to the modular description of control flow that is discussed in subsection 4.3.6 and 7.2.3. In the case of Racket the work was initiated by the fact that multiple usages of `call/cc` would interfere with each other. Flatt et al. made use of dynamic-winding and continuation marks to implement a solution to this problem [21, 22]. These continuation marks provide a system that almost equivalent with the labeled continuations of the Frame VM. In fact, their implementation is very similar to the scoped control graph-approach discussed in chapter 9. The caching strategy they use makes their system behave more closely to the current implementation of control frames.

Besides the Scheme community, other people have also looked in to execution of programming languages using continuations. Tismer constructed a version of Python with continuation support. This allowed for a better implementation of generators and co-routines [49]. Other researchers looked at supporting continuations in LLVM. This allows to easily target various architectures using a single compiler, in their case one for Parallel ML [17].

8.2 Semantics Frameworks

Dynamix is a DSL that allows one to write semantics of programming languages and to execute programs using these semantics. That makes Dynamix an executable semantics framework. Other frameworks in this category are, among others, K [37], PLT-Redex [18], CBS [9] or Dynsem [51, 50].

Most of these frameworks differ from Dynamix in that they also include other tools required for program execution, like parsing. For Dynamix these other features are provided by its integration with the Spoofax language workbench. This has the benefit that any improvements made to the components of this workbench directly improve working with Dynamix.

In addition to being executable semantics, a Dynamix specification, when combined with the Frame VM, also has the benefit of modularity with respect to control. This is something other semantic description languages often have problems with as they leave all aspects of the semantics explicit, even when not locally used. This has the effect that a change to a component of the semantics might require changes to other components that did not use the changed component. Modular semantics frameworks and implicit propagation have been used in to solve this problem. Examples are funcons [10] as composable language components or its underlying semantic framework I-MSOS [34].

8.3 Meta-Programming

Dynamix is, besides a semantics framework also a meta-programming language. More specifically Dynamix is a runtime, manual and heterogeneous meta-programming language, using the taxonomy created by Sheard [41]. This means that the bytecode programs created by execution of a Dynamix specification can be ran directly (runtime meta-programming), quoted blocks and anti-quotation have to be placed manually to annotate stage changes (manual meta-programming) and the meta- and object-language (Dynamix and Roger respectively) are two different languages (heterogeneous meta-programming).

Many other meta-programming languages do exist, for example Template Haskell [43] or Meta-ML [42, 47]. Dynamix took inspiration for its quoted block notation from the latter language.

Chapter 9

Future Work

Despite the fact that a lot of work has been done during this thesis, there are plenty of interesting next steps to take. In this chapter some of these future steps are described.

9.1 Refine Control Frames

In chapter 3 the notion of control frames was introduced. These form the building blocks that the Frame VM uses for creating control flow. While they can be used to model a lot of different types of control flow, there is still room for improvement. For example the current design of control frames is not able to model delimited continuations without using a small hack to create a control frame-pointer.

An other area for improvement is the internal memory of a control frame. Using the scopes-as-frames paradigm it is possible to have a nice description of the runtime memory layout, but register values just exist. These register values represent intermediate values for computations that will, eventually, result in updates of values that are present in the scope graph of a program. In the current model, these values are magically created when needed. They are however a vital part of the data flow of a program. Therefore a good formalism has to be found that can represent this aspect of program execution.

There is also still the question if control frames are the building blocks that can model *all* types of control flow. They seem to be close by the fact that many types can be modeled using them. However, there are a lot of different control-operators in the literature (e.g. call/cc, shift, reset, control) [23]. If control frames really are the basic building blocks, it should be possible to express these control-operators using them. But maybe there is an even more fundamental construct that could be used to model the control flow of a program.

9.1.1 Delimited Continuations

In contrast to continuations, delimited continuations only capture the rest of the program up to a certain point. This point that delimits the continuation is a prompt [19]. In the Scheme-like code in Figure 9.1, `(# ..)` is the prompt that delimits the continuation. Without considering the last line of this program, the output is exactly what is to be expected from a normal application of call/cc. The prompt acts transparently and the captured continuation is just the following expression with a hole `(+ 1 (+ 2 • ))`. In the case of a delimited continuation, the prompt aborts execution of the current continuation and calls back to the original invocation of of the delimited continuation. This makes that the delimited continuation can be seen as the expression `(x (+ 2 •))`, where `x` is the continuation of the code that invoked the delimited continuation. Therefore the output of the last line is 42, as `x` is conceptually `(+ 39 •)` and the value that plugs the hole in the delimited continuation is 1.

```
(define x #f)
(+ 1
  (#
    (+ 2 (call/cc
      (lambda
        (k)
        (set! x k)
        3
      )
    ))
  )
)                   ; 6
(+39 (k 1))         ; 42
```

Figure 9.1: An example of delimited continuations in a Scheme-like language

Compiling this to the Frame VM is currently not possible, but a method was investigated on how this could be done. Intuitively, the prompt `(# ..)` feels like some sort of checkpoint that gets to decide what to do next. With this observation made, the prompt can be translated to a control frame with a continuation stored in the continuation slots for what to do next. Initially this continuation points to the previous current control frame in order to make it act transparently. The only thing that executing code in this control frame does is to invoke this continuation.

While this solution already works for the first line of the program shown in Figure 9.1, this implementation does not suffice for the full program. For this solution to work, the continuation stored in the prompt control frame `#` has to be updated to point to the correct continuation. This means that on invocation of a delimited continuation, the continuation of the caller has to be stored in the continuation slot of the prompt.

To be able to update this value, a reference is needed to the control frame of the prompt. As a small hack a continuation to this control frame can be used to get this reference (Figure 9.2). This is not a proper reference (as these do not exist in the current design), but as it can be used as one it will work for the sake of explanation. Using this pointer to the prompt control frame, it becomes easy to update the continuation of that control frame on the invocation of a delimited continuation.

There is however still a case where this construction does not suffice. Consider two instances of a delimited continuation. When these both share the same prompt, it could be possible that one updates the prompt target while the other was still relying on its old value. To remedy this possibility, it is needed to create copies of control frames before the prompt target is updated. By creating this copy, it is ensured that a each usage of the prompt has its

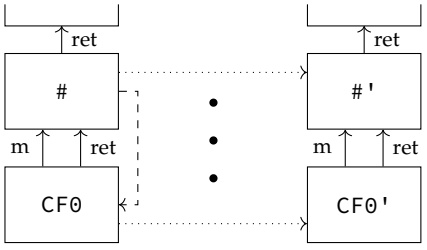


Figure 9.2: Modeling delimited continuations using control frames. On the right the copied sub-graph is shown.

```
def foo():
  return bar()

def bar()
  return 42

foo()
```

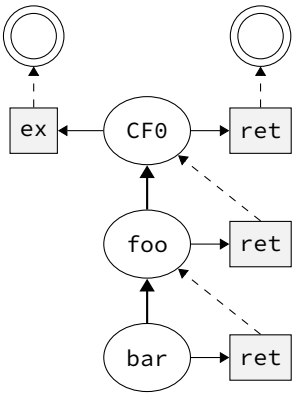


Figure 9.3: A scoped control graph. Each return address of a function shadows the return address of its caller, but the exception handler is shared between all calls.

own control frame subgraph with respect to the prompt target. This eliminates the possibility of overwriting the prompt target of some other part of the program.

Scoped Control Graphs

In the example from Figure 9.2, the slot `m` was used to hold a pointer to the prompt control frame. As said when introducing this notation, this is a bit of a hack as these pointers are not proper continuations. Furthermore, the fact that it needs to be a pointer at all is a sign of a shortcoming of the model.

If it was a proper continuation, this would mean that any child control frame would copy this continuation to its on continuation slots. When the value then needs to be updated, it has to be updated in all control frames that copied it to themselves. By making it a pointer, the copy will only perform a pointer copy and an update updates the value at the pointer location. As an effect, the update only has to be performed once at the pointer location.

The problem is that the current model of continuations stored in control frames does not consider pointers to control frames. Furthermore the implicit copy of unchanged continuations to newly created control frames always felt a bit off. With these two problems in mind, there has to be a better formalization of the model. This new model is not thoroughly investigated yet, but seems to at least improve things and go in the right direction.

The key observation of the new model is that continuations can shadow each other. For example, a return address of a function shadows the return address of the caller of the function. With this in mind, inspiration can be drawn from scope graphs and the way these handle shadowing of names. Here a declaration is defined in a single scope and other scopes with this first scope as (transitive) parent can reference this declaration. By adding the property that resolution of a reference always resolves to the closest declaration in the graph, shadowing of declarations is achieved.

Translating this to a model with continuation slots and control frames, gives a system where each control frame is a scope and each continuation slot is a declaration. This declaration can be in either the current control frame or reachable via a relation to parent control frames. The control frames captured in a continuation can now be seen as associated control frames of a continuation declaration. Figure 9.3 shows this new scoped variant of the control graph, using a similar drawing style as scope graphs.

9.2 Frame VM Improvements

Functionally the implementation of the Frame VM is complete enough to compile various programming languages to it (see chapter 5, and 7). There are however a number of features that need to be added to make it able to encode more language features (memory copy), provide better debugging support (loci of control, better error handling) and most importantly execute programs at a decent speed.

To speed up the execution of the bytecode, the VM needs to be re-implemented as an actual VM. There are a number of potential paths that have to be investigated for this implementation. An implementation on the JVM provides good integration with the Spoofax language workbench. The JVM is however not designed to work with continuations, making such an implementation hard. Using PyPy might be a better bet, as an existing Racket implementation in this framework proved that working with continuations is feasible in PyPy [8].

A second path to speed up the execution is to transpile the bytecode to a different target language. Potential targets are LLVM bytecode, C- - or Schez-Scheme. Schez-Scheme is probably the easiest target, as it is designed from the ground up to work with continuations and still provides good performance [16]. Farvardin and Reppy [17] have created an encoding to add proper continuations to LLVM, so this is a feasible target as well.

9.2.1 Bytecode Optimizations

If you take a quick look at the bytecode generated by the Dynamix meta-compiler, it is clear that this code is far from efficient. The Dynamix compiler assumes an infinite set of registers, and as an effect will never reuse registers that are no longer in use. In addition, the Dynamix compiler uses a lot of extra registers, that are not strictly necessary at first glance. This is done to make sure that no code gets executed twice. Therefore new registers are created for most intermediate values. However, it is not necessary in most cases to use all of these extra registers as registers can often be reused. Furthermore all these extra registers make the execution of continuation calls more expensive. Reducing the amount of registers that are used can therefore greatly increase the execution speed of the VM

A first step in reducing the amount of registers could be to try to inline the expressions into expression trees. These trees could then easily be decomposed into a list of instructions (without expression trees), by using a small set of accumulator registers. Reusing these accumulators between different expression trees is now trivial as there is no overlap between their executions. After this relatively simple step to reduce the number of registers used, it becomes a lot harder really quick. This is because the register allocation problem is an NP-Complete problem [2].

An other instance where more code is generated than necessary is in the case that data frames are created and destroyed without any operation in between that actually uses that frame. The creation of frames with a size of zero are often an indication of this behavior. These empty data frames can often be omitted as long as all path traversals through this frame are updated to mirror the deletion of this step in their resolutions. They can however not be removed from the Dynamix specification, as this would break the scopes-as-frames paradigm.

Performing these types of optimizations could probably be done at compile time. The fact that the layout of these frames, and the resolution paths through them, is statically known using the scope graph, makes that we can statically simplify this graph and apply the same simplification to the generated bytecode. This is currently hard to do as this relation between the scope graph and the data frames is not checked by the Dynamix type system. But if we ensure that the layouts are the same and every scoping related bytecode instruction can be related to its position in the scope graph, these types of optimizations are possible. A simpler solution is to do simple checks for patterns in the generated bytecode and apply

some transformation to the code, similar to what Crielaard and Beinema did in their Rust compiler [11].

9.2.2 Frame VM Error Handling

When the Frame VM cannot execute a given instruction, for example getting slot 1 from a data frame with size 0 or from a piece of data like an integer which is not even a frame, the VM simply halts execution (crashes). A more practical solution would be to raise an exception in the program that is currently executing. However, this poses a number of challenges that make it hard to implement correctly.

The most basic case where this does not work in a straightforward way is when the target language has no notion of exceptions. This simply means that there is no exception handler available to invoke (other than the root exception handler, but this still halts the entire execution). A compiler designer can however use this exception to handle common exceptions like null-pointers or division by zero by handling them locally without exposing them to the entire language.

But even assuming there is some exception handler that could be used, there is still the issue of what this exception should be. Maybe a string containing a message and a reference to a control frame that could be used to reconstruct a stack trace? This does, however, not work for a language without strings, or a language with a string encoding that differs from the encoding the machine uses in the exception it raises.

It should by now be clear that such an universal solution is never going to work in practice. A workable solution could be to invoke the closest exception handler and use a type system for Dynamix to enforce that certain operations provide an exception handler. This handler would then handle the exception in a way that fits the implemented language. It could still terminate execution, could convert the exception to an exception that is valid in the implemented language or return some default value. A disadvantage of this approach is that a lot of instructions have to be wrapped, even though they might never raise an exception. Again, a hypothetical type system for Dynamix might be able to use information from the scope graph (for example the good-heap property) and the data flow to know when certain frame operations are guaranteed to not fail.

9.2.3 Memory Copy

When using the scopes-as-frames paradigm, your memory layout resembles a directed graph. This has implications for the way a memory copy operation works. In the case of linear memory, creating a copy of a section of memory is simple. Given a pointer to the start of a memory region, a pointer to the destination and the length of the region, you can simply loop over the memory region and copy each of the values to the destination [24]. In the case of graph-based memory layouts, defining this region in a similar way is not sufficient. Instead, we need to define the memory region as a subgraph of the original graph.

Performing the subgraph copy is already possible in Roger, but it is a very tedious task. To aid in this task, Dynamix should provide functionalities that make the definition of the subgraph simpler and generate the required copy operations automatically.

9.3 Dynamix

Besides the VM itself, Dynamix also has plenty of room for expansion and improvements. For example, the set of library functions that was described in section 6.3 is currently very incomplete. This library has to be greatly expanded to allow a language designer to cobble a language together by combining predefined semantic components. Furthermore, this library is currently not integrated in the Dynamix runtime and has to be added manually to each

project by copy-pasting a file. For a better workflow, this library has to be shared between different language implementations.

Currently it is not possible to write a Dynamix specification for a language using Statix for the name binding analysis. A second improvement would therefore be to update Dynamix to work with (both Nabl2 and) Statix.

9.3.1 Type System

In the earliest versions of Dynamix a small type system was implemented. This type system would check variable references and would make sure that every instance of a given rule had the same type. For this check multiple different types were distinguished. Values could be AST elements, runtime values or lists of compiled instructions.

These three different types proved to be too general to get adequate type errors in a language implementation in Dynamix. For example, the Dynamix primitive `explode-string` takes a string (AST element type) and `resolve` will produce a memory lookup path (also of the AST type). Giving the result of the latter as an argument to `explode-string` would clearly result in a runtime error, but the type system was unable to detect this error. As implementing many more (sub)types was too much work for the available time and as the existing type system prevented quick development of the at that point still evolving language, it was decided to limit the type system to simple name analysis within a rule instance.

A powerful type system could however provide a lot of benefits to a language designer. It could for example crosscheck the Dynamix specification with the syntax definition (and desugaring steps) of a language to discover missed cases. Furthermore adherence to the scope graph can be checked. This would check if every Dynamix rule that needs to create a scope according to the scoping rules will do so and that every variable reference happens from the correct scope. Implementing this would be hard and requires updates to the available API for querying the scope graph, but could provide partial memory-safety guarantees following the good-heap property [3].

9.3.2 Debugging

When debugging a program, it is needed to trace the execution point in the compiled code back to some statement in the source language. When using a Dynamix meta-compiler, the programmer needs to trace back the steps of the compiler by hand, in order to figure out where the error occurred. This is done by comparing the generated bytecode and the Dynamix specification of the compiled language. This is not only a lot of work, but most end users won't even have a copy of this specification at hand.

In order to make this tracing an automated task, the debugger must be able to identify the *locus of control* and map the different loci to source-code locations [26]. This requires to either have a really smart decompiler (which in the case of Dynamix has to be a meta-decompiler) or the generated code has to be instrumented to help a debugger. GCC uses this second approach and inserts DWARF debugging information in the generated binaries [31].

The full power of DWARF is however not needed in the case of the Frame VM. This is because many of the DWARF features are already covered by scopes-as-frames and control frames. It is already possible to retrace the call stack using the return continuations stored in the current control frame, or to get information about heap-allocated objects (data frames). The only missing step is the mapping from generated- to source-code that provides the loci of control.

Chapter 10

Conclusion

The goal of this thesis was to find an instruction set for a VM that uses a language independent memory abstraction and supports many control flow constructs. This would allow for the execution of many wildly different languages on a single platform. As an effect, the VM would help in the rapid development of programming languages in, for example, a language workbench.

For the language-independent description of runtime memory the scopes-as-frames paradigm was used. This dictated a memory system consisting of linked data frames storing the execution data. Memory lookups in this system are in a one-to-one correspondence to the resolution paths of a reference in a scope graph, aiding in code generation.

To describe the control flow of the execution of a program, control frames were introduced. Incorporating ideas from CPS and call frames, these control frames store continuations, the current PC and local (temporary) memory. This allows them to be used as the building blocks of various types of control flow.

Together with data frames, control frames form the core data structures used by the Frame VM. Usage of these components to encode different programs showed the operations that were needed to make effective use out of them. These operations became the core of the instruction set that we were looking for in this thesis. This instruction set was then captured in the bytecode language of the Frame VM, Roger.

Investigating the completeness of this instruction set was done by constructing compilers for various languages. Scheme was used for its inclusion of call/cc and lambdas and Tiger was used as a language with normal functions. Furthermore other students compiled Prolog and Rust to the Frame VM to test its instruction set.

However, until Dynamix was created, writing these compilers was still a tedious task. The Dynamix language introduced in this thesis is an executable semantics description language. This allows for a declarative semantics description of a language, with which a program can be meta-compiled directly to the Frame VM.

Using Tiger and Scheme as a test, Dynamix specifications were created that describe these languages. The resulting semantics specifications were then used to compile various programs. In the case of Scheme, examples of call/cc usage were tested and for Tiger an existing suite of benchmark programs was ran.

In addition to these examples, the modularity of the Dynamix specifications with respect to control flow was shown. To show this, the Tiger language was extended with exception handling and generator functions. These new constructs introduced new Dynamix rules for each of them, but doing so did not require any modifications to the existing language specification. This showed that changes can be made to the control flow of a language without effecting other parts of the semantics that did not interfere with these changes.

While the execution speed of the Frame VM was still poor, these examples did show the applicability of both Dynamix and the Frame VM. There are however multiple areas in

which the system developed during this thesis can be improved. For example, control frames as they are now cannot fully describe delimited continuations and a Dynamix specification of a language does not offer any guarantees about the execution of programs. Future work is therefore needed in these areas. In chapter 9 various directions were given for this, including a scoped version of control frames, and components for a type system for Dynamix.

Bibliography

[1] Harold Abelson et al. "Revised 4 report on the algorithmic language scheme". In: *ACM SIGPLAN Lisp Pointers* 4.3 (1991), pp. 1–55.

[2] Alfred V Aho and Stephen C Johnson. "Optimal code generation for expression trees". In: *Journal of the ACM (JACM)* 23.3 (1976), pp. 488–501.

[3] Hendrik van Antwerpen et al. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 114.

[4] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[5] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[6] Andrew W. Appel. *Tiger testcases*. [Online; accessed 14-October-2019]. 1996. URL: https://www.cs.princeton.edu/~appel/modern/testcases/.

[7] Casper Bach Poulsen et al. "Scopes describe frames: A uniform model for memory layout in dynamic semantics". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[8] Spenser Bauman et al. "Pycket: a tracing JIT for a functional language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 22–34.

[9] L Thomas van Binsbergen, Peter D Mosses, and Neil Sculthorpe. "Executable component-based semantics". In: *Journal of logical and algebraic methods in programming* 103 (2019), pp. 184–212.

[10] Martin Churchill et al. "Reusable components of semantic specifications". In: *Transactions on Aspect-Oriented Software Development XII*. Springer, 2015, pp. 132–179.

[11] Bram Crielaard and Emiel Beinema. "Compiling Rust to the FrameVM". In: (2019).

[12] Olivier Danvy. "On evaluation contexts, continuations, and the rest of the computation". In: *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College*. 2004, pp. 13–23.

[13] Brian Davis et al. "The case for virtual register machines". In: *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. 2003, pp. 41–49.

[14] Akim Demaille and Roland Levillain. *Tiger Compiler Reference Manual*. Jan. 2018.

[15] Edsger W Dijkstra. "Recursive programming". In: *Numerische Mathematik* 2.1 (1960), pp. 312–318.

[16] R Kent Dybvig. "Three implementation models for scheme". PhD thesis. University of North Carolina at Chapel Hill, 1987.

[17] Kavon Farvardin and John Reppy. "Compiling with Continuations and LLVM". In: *arXiv preprint arXiv:1805.08842* (2018).

[18] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[19] Mattias Felleisen. "The theory and practice of first-class prompts". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 180–190.

[20] Matthew Flatt et al. *The Racket Reference*. 2014.

[21] Matthew Flatt et al. "Adding delimited and composable control to a production programming environment". In: *ACM SIGPLAN Notices* 42.9 (2007), pp. 165–176.

[22] Matthew Flatt et al. "Rebuilding racket on chez scheme (experience report)". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–15.

[23] Martin Gasbichler and Michael Sperber. "Final shift for call/cc: direct implementation of shift and reset". In: *ACM SIGPLAN Notices* 37.9 (2002), pp. 271–282.

[24] GCC. *gcc/libgcc/memcpy.c*. [Online; accessed 5-September-2019]. Nov. 2011. URL: https://github.com/gcc-mirror/gcc/blob/master/libgcc/memcpy.c.

[25] Chris Hanson and Massachusetts Institute of Technology. Scheme Team. *MIT/GNU Scheme Reference Manual*. Free Software Foundation, 2019.

[26] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. "C-—: A portable assembly language that supports garbage collection". In: *International Conference on Principles and Practice of Declarative Programming*. Springer. 1999, pp. 1–28.

[27] Lennart CL Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *ACM sigplan notices* 45.10 (2010), pp. 444–463.

[28] Richard Kelsey, William Clinger, Jonathan Rees, et al. "Revised 5 report on the algorithmic language Scheme". In: (1998).

[29] Andrew Koenig and Bjarne Stroustrup. "Exception handling for C++". In: *Proceedings of Usenix*. Vol. 90. 1989, pp. 149–176.

[30] Stefan Marr. *are-we-fast-yet*. [Online; accessed 14-October-2019]. 2019. URL: https://github.com/smarr/are-we-fast-yet.

[31] Arnaldo Carvalho de Melo. "The 7 dwarves: debugging information beyond gdb". In: *Proceedings of the Linux Symposium*. Citeseer. 2007.

[32] Metaborg. *Stratego API*. [Online; accessed 12-December-2019]. Dec. 2019. URL: http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/stratego-api.html.

[33] Luka Miljak. "Building a Compiler from Prolog to FrameVM". In: (2019).

[34] Peter D Mosses and Mark J New. "Implicit propagation in structural operational semantics". In: *Electronic Notes in Theoretical Computer Science* 229.4 (2009), pp. 49–66.

[35] Pierre Neron et al. "A theory of name resolution". In: *European Symposium on Programming Languages and Systems*. Springer. 2015, pp. 205–231.

[36] John C Reynolds. "The discoveries of continuations". In: *Lisp and symbolic computation* 6.3-4 (1993), pp. 233–247.

[37] Grigore Roșu and Traian Florin Șerbănuță. "An overview of the K semantic framework". In: *The Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434.

[38] Amr Sabry and Matthias Felleisen. *Is continuation-passing useful for data flow analysis?* Vol. 29. 6. ACM, 1994.

[39] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. *PEP 255 – Simple Generators*. [Online; accessed 1-November-2019]. June 2001. URL: https://www.python.org/dev/peps/pep-0255/.

[40] Robert W Sebesta. *Concepts of programming languages*. Boston: Pearson, 2012.

[41] Tim Sheard. "Accomplishments and research challenges in meta-programming". In: *International Workshop on Semantics, Applications, and Implementation of Program Generation*. Springer. 2001, pp. 2–44.

[42] Tim Sheard. "Using MetaML: A staged programming language". In: *International School on Advanced Functional Programming*. Springer. 1998, pp. 207–239.

[43] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. 2002, pp. 1–16.

[44] Yannis Smaragdakis. "Next-paradigm programming languages: what will they look like and what changes will they bring?" In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 187–197.

[45] Michael Sperber et al. "Revised 6 report on the algorithmic language Scheme". In: *Journal of Functional Programming* 19.S1 (2009), pp. 1–301.

[46] Christopher Strachey. "Continuations: A mathematical semantics for handling full jumps". In: (1974).

[47] Walid Taha and Tim Sheard. "MetaML and multi-stage programming with explicit annotations". In: *Theoretical computer science* 248.1-2 (2000), pp. 211–242.

[48] Robert D. Tennent. "The denotational semantics of programming languages". In: *Communications of the ACM* 19.8 (1976), pp. 437–453.

[49] Christian Tismer. "Continuations and stackless Python". In: *Proceedings of the 8th international python conference*. Vol. 1. 2000.

[50] Vlad Vergu, Pierre Néron, and Eelco Visser. "DynSem: A DSL for dynamic semantics specification". In: *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[51] Vlad Vergu, Andrew Tolmach, and Eelco Visser. "Scopes and Frames Improve Meta-Interpreter Specialization". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[52] David A Watt. *Programming language design concepts*. John Wiley & Sons, 2004.

Acronyms

AST abstract syntax tree

PC program counter

REPL read-evaluate-print loop

CPS continuation passing style

VM virtual machine

DSL Domain Specific Language

LEP Language Engineering Project

DSL Domain Specific Language

IDE Integrated Development Environment

CFG Control Flow Graph

Glossary

continuation A continuation describes the evaluation of a program from its current state to its final (terminal) state, including its semantic context (i.e., the environment and store) [46] . viii, 11–27, 31, 32, 34–39, 45, 50, 52–55, 58–60, 62–64, 66–69, 73, 79–82, 84, 85, 87, 88, 91–96, 98, 99, 105

control flow The execution order of instructions in a program. i, vii, 2, 3, 7, 11–17, 21–23, 38, 41, 42, 45, 47, 49, 52, 53, 58, 79, 83, 85, 91–93, 99, 107

control frame An abstraction of program control introduced in section 3.3. i, viii, ix, 3, 7, 11, 15–18, 21–27, 30–39, 45, 47, 50, 53–56, 85, 88, 91–95, 98–100, 107, 115

control graph A graph consisting of control frames. The graph is used for visualizing control flow constructs . vii, viii, 16, 17, 95, 107

data frame Heap-allocated frames used by the Frame VM for storing data. See chapter 2 how they are created from a scope graph using the scopes-as-frames paradigm. vii, ix, 7, 8, 21–25, 28–32, 34, 36–39, 42–44, 51, 52, 54, 55, 80, 81, 85, 91, 97–99, 107, 113

Frame VM The VM created during this thesis. It uses control frames and data frames to model control flow and memory respectively. Its full design is described in chapter 4. i, ii, vi–ix, 2, 3, 19–22, 25, 27, 29–33, 36, 41–45, 47–49, 51, 54, 56–60, 62, 74, 76, 77, 79, 82, 84, 85, 91–94, 96–99, 107, 113, 114

Roger The bytecode language used by the Frame VM. See chapter 4 for its syntax and semantics. vii–ix, 19–24, 26–32, 39, 43, 48, 58, 59, 62, 73, 84, 91, 92, 97, 99, 109, 112

scope graph A graph-representation of the naming structure of a program introduced by Neron et al. [35]. vii, viii, 6–9, 11, 15, 16, 23, 44, 49, 51, 55, 57, 60, 62, 74–76, 93, 95, 97–99, 107

scoped control graph An adaption of a control graph where continuations are properly scoped instead of using the implicit copy technique. 92

scopes-as-frames A paradigm described by Bach Poulsen et al. that describes the relation between the scope graph of a program and its memory layout [7]. i, vii, 2, 6–8, 11, 15, 19, 20, 22, 47, 51, 84, 91, 93, 96–99, 107

Spoofax language workbench A language workbench for development of textual domain-specific languages and their IDEs [27]. 2, 41, 44, 47, 57, 58, 92, 96

Appendix A

Additional Frame VM Syntax and Semantics

i	::=	**debug**()	instruction list	e	::=	**negi**(e)	expression
	|	**debug!**()		|	**muli**(e , e)	
	|	**tick**()		|	**subi**(e , e)	
	|	**tock**()		|	**divi**(e , e)	
	|	**forceGC**()		|	**modi**(e , e)	
	|	**print**(e)		|	**lti**(e , e)	
	|	**set**(e , e , e)		|	**gti**(e , e)	
	|	**set**(p , e)		|	**ori**(e , e)	
	|	**set**(e , e)		|	**xori**(e , e)	
	|	**setC**(e , e , e)		|	**andi**(e , e)	
	|	**setC**(c , e)		|	**int?**(e)	
	|	**setC**(e , e)		|	**cont?**(e)	
	|	**newscope**(e , k)		|	**frame?**(e)	
	|	**exitscope**(p)		|	**CF?**(e)	
j	::=	**return**(d)	control instr	|	**code?**(e)	
	|	**yield**(d , e)		|	**new**()	
	|	**call**(e , e , e)		|	**size**(e)	
	|	**call**(e , e)		|	**get**(e)	
	|	**tailcall**(e)		|	**get**(p)	
	|	**tailcall**(e , e)		|	**get**(e , e)	
	|	**try**(e , e , e , e , e)		|	**getC**(e)	
	|	**try**(e , e , e)		|	**getC**(c)	
	|	**throw**(e)		|	**getC**(e , e)	
	|	**newscope**(e , k , e)		|	**sload**(" h+ ")	
	|	**exitscope**(p , e)		|	**cload**(' h ')	

$h \in$ Character

Figure A.1: Additional Roger syntax, extends the syntax from Figure 4.2.

set(p, e)	⇒	**set**(**getcurrent**( ), p, e)	
set(e1, e2)	⇒	**set**(**getcurrent**( ), e1, e2)	
set(e1, [n], e2)	⇒	**set**(e1, **iload**(n), e2)	*Replaces* `I-Set` (*Figure 4.17*)
setC(c, e)	⇒	**setC**(**curCF**( ), c, e)	
setC(e1, e2)	⇒	**setC**(**curCF**( ), e1, e2)	
get(p)	⇒	**get**(**getcurrent**( ), p)	
get(e)	⇒	**get**(**getcurrent**( ), e)	
get(e, [n])	⇒	**get**(e, **iload**(n))	*Replaces* `E-GetSlot` (*Figure 4.22*)
getC(c)	⇒	**getC**(**curCF**( ), c)	
getC(e)	⇒	**getC**(**curCF**( ), e)	
newscope(e, k)	⇒	f ← **new**(e) **link**(f, k) **mkcurrent**(f)	
newscope(e1, k, e2)	⇒	**newscope**(e1, k) **jump**(e2)	
exitscope(p)	⇒	**mkcurrent**(**get**(p))	
exitscope(p, e)	⇒	**exitscope**(p) **jump**(e)	
return($\overline{e}$)	⇒	**callC**(**getC**(**iload**(0)), $\overline{e}$)	
throw(e)	⇒	**callC**(**getC**(**iload**(1)), e)	
new( )	⇒	**new**(**iload**(0))	
cload(h)	⇒	**iload**(n)	$n =$ UTF-16 integer value of h
f ← **sload**($\overline{h}^{i \in [0..n]}$)	⇒	f ← **new**(**iload**($n+2$)) **set**(f, 0, $n+1$) **set**(f, 1, $\overline{m}^0$) ⋮ **set**(f, $n+1$, $\overline{m}^n$)	$\overline{m}^x =$ UTF-16 integer value of $\overline{h}^x$

Figure A.2: Desugaring rules of many syntax constructs from Figure A.1.

yield($\overline{e1}^{i\in[0..n]}$, e2)	⇒	**callC**(**getC**(**iload**(0)), $\overline{e1}^0$, $\cdots$, $\overline{e1}^n$, **curC**(e2))
call(e1, e2, e3)	⇒	cf ← **newCF**(e1) **setC**(cf, **iload**(0), **curC**(e3)) **callCF**(cf, e2)
call(e1, e2)	⇒	cf ← **unpackC**(e1) **setC**(cf, **iload**(0), **curC**(e2)) **callC**(e1)
tailcall(e1, e2)	⇒	cf ← **newCF**(e1) **setC**(cf, **iload**(0), **getC**(**iload**(0))) **callCF**(cf, e2)
tailcall(e)	⇒	cf ← **unpackC**(e) **setC**(cf, **iload**(0), **getC**(**iload**(0))) **callC**(e)
try(e1, e2, e3, e4, e5)	⇒	c_try ← **newC**(**newCF**(e1), e2) c_catch ← **newC**(**newCF**(e3), e4) **try**(c_try, c_catch, e5)
try(e1, e2, e3)	⇒	cf_try ← **unpackC**(e1) c_next ← **curC**(e3) **setC**(cf_try, **iload**(1), e2) **setC**(cf_try, **iload**(2), c_next) **setC**(**unpackC**(e2), **iload**(2), c_next) **callC**(e1)

Figure A.3: Desugaring rules of many syntax constructs from Figure A.1 (cont.).

$$\boxed{B, X \vdash \langle i, h, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{i}} \langle h, \gamma, o\rangle}$$

I-SetIndex

$$\frac{\begin{array}{c} R_{h_1}(\chi) = \rho \qquad \rho(r_1) = \texttt{Frame}(f) \\ \rho(r_2) = \texttt{IntV}(n) \qquad \rho(r_3) = v \qquad s \in Dom(D_{h_1}(\chi)) \qquad D_{update}(\chi, n, v)_{h_1} = h_2 \end{array}}{B, X \vdash \langle \textbf{set}(r_1, r_2, r_3), h_1, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{i}} \langle h_2, \gamma, o\rangle}$$

I-SetContinuationIndex

$$\frac{\begin{array}{c} R_{h_1}(\chi_1) = \rho \qquad \rho(r_1) = \texttt{ControlFrame}(\chi_2) \\ \rho(r_2) = \texttt{IntV}(n) \qquad \rho(r_3) = k \qquad k = \texttt{Continuation}(\chi_2, l) \qquad C_{update}(\chi, n, k)_{h_1} = h_2 \end{array}}{B, X \vdash \langle \textbf{setC}(r_1, r_2, r_3), h_1, \chi, \gamma, o\rangle \xrightarrow{\mathfrak{i}} \langle h_2, \gamma, o\rangle}$$

$$\boxed{B, X \vdash \langle e, \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-Size

$$\frac{\rho(r_1) = \texttt{Frame}(f) \qquad D_h(f) = \sigma \qquad v = \texttt{IntV}(|\sigma|)}{B, X \vdash \langle \textbf{size}(r_1), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

E-NegInt

$$\frac{\rho(r_1) = \texttt{IntV}(z) \qquad v = \texttt{IntV}(-z)}{B, X \vdash \langle \textbf{negi}(r_1), \rho, h, \chi, \gamma\rangle \xrightarrow{\mathfrak{e}} \langle v, h, \gamma\rangle}$$

Figure A.4: Additional semantic rules for Roger.

The semantic rules for the following expressions are left out, as they can easily be deduced from other semantic rules:

- `muli`, `subi`, `divi` and `modi` are similar to `E-ADDINT` (Figure 4.19)
- `lti`, `gti`, `ori`, `xori` and `andi` are similar to `E-EqualInt-True` / `False` (Figure 4.20)
- `int?`, `cont?`, `frame?`, `CF?` and `code?` are similar to `E-IsNull-True` / `False` (Figure 4.20)

Semantic rules for the debug instructions are only described informally:

- `print` prints raw data values to the output stream (e.g. `IntV( 42 )`, `Frame( frame_2 )`).
- `forceGC` forces the garbage collector to run, blocks until completion.
- `tick` and `tock` start and stop an internal timer. On `tock` prints the time since the last `tick` to the output stream.
- `debug` converts the internal machine state to a DOT-graph. When the machine terminates the output stream is replaced by the most recent debug output.
- `debug!` converts the internal machine state to a DOT-graph, terminates execution and replaces the output stream by the debug output.

Appendix B

Dynamix Primitive Frame VM Operations

In section 6.5 the semantics of Dynamix were introduced. For simplicity a lot of the primitive operations were not shown in that chapter. Instead, the semantics of these primitive operations are included in this appendix.

$$
\begin{array}{lcl}
A \vdash \langle \texttt{new}, [e1] \rangle & \xrightarrow{\text{p}} & \langle A, [\,], [\![\ \texttt{RGR\_New}\ , [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{size}, [e1] \rangle & \xrightarrow{\text{p}} & \langle A, [\,], [\![\ \texttt{RGR\_FSize}\ , [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{get}, [e1, e2] \rangle & \xrightarrow{\text{p}} & \langle A, [\,], [\![\ \texttt{RGR\_Get}\ , [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{link}, [e1, e2, e3] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_Link}\ , [e1, e2, e3]]\!]^{\perp}], \perp \rangle \\
A \vdash \langle \texttt{set}, [e1, e2, e3] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_Set}\ , [e1, e2, e3]]\!]^{\perp}], \perp \rangle \\
A \vdash \langle \texttt{cur}, [\,] \rangle & \xrightarrow{\text{p}} & \left\langle [\,], [\![\ \texttt{RGR\_ScopeGetCurrent}\ , [\,]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{mkcur}, [e1] \rangle & \xrightarrow{\text{p}} & \left\langle A, [[\![\ \texttt{RGR\_ScopeSetCurrent}\ , [e1]]\!]^{\perp}], \perp \right\rangle
\end{array}
$$

Figure B.1: Semantics of Dynamix primitives for data frame operations

$$
\begin{array}{lcl}
A \vdash \langle \texttt{jump}, [e1] \rangle & \xrightarrow{\text{p}} & \left\langle A, [[\![\ \texttt{RGR\_Jump}\ , [e1]]\!]^{\perp}], \perp \right\rangle \\
A \vdash \langle \texttt{jumpz}, [e1, e2, e3] \rangle & \xrightarrow{\text{p}} & \left\langle A, [[\![\ \texttt{RGR\_JumpZ}\ , [e1, e2, e3]]\!]^{\perp}], \perp \right\rangle \\
A \vdash \langle \texttt{callC}, [e1] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_ContCall}\ , [e1]]\!]^{\perp}], \perp \rangle \\
A \vdash \langle \texttt{callC}, [e1|t] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_ContCallWith}\ , [e1, t]]\!]^{\perp}], \perp \rangle \\
A \vdash \langle \texttt{callCF}, [e1, lbl] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_CFCall}\ , [e1, lbl]]\!]^{\perp}], \perp \rangle \\
A \vdash \langle \texttt{returnCF}, [e1] \rangle & \xrightarrow{\text{p}} & \langle A, [[\![\ \texttt{RGR\_CFReturn}\ , [e1]]\!]^{\perp}], \perp \rangle
\end{array}
$$

Figure B.2: Semantics of Dynamix primitives for control operations

$$
\begin{array}{lcll}
A \vdash \langle \texttt{null}, [\,] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_NLoad}, [\,]]\!]^{\perp} \rangle & \\
A \vdash \langle \texttt{char}, [c] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_CharLoad}, [c]]\!]^{\perp} \rangle & \\
A \vdash \langle \texttt{int}, [z] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_ILoad}, [z]]\!]^{\perp}) \rangle & \\
A \vdash \langle \texttt{int}, [s] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_ILoad}, [z]]\!]^{\perp}) \rangle & \text{where } z = \text{integer value of } s \\
 & & & \\
A \vdash \langle \texttt{printc}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [[\![\texttt{RGR\_PrintChar}, [e1]]\!]^{\perp}], \perp \rangle & \\
A \vdash \langle \texttt{print}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [[\![\texttt{RGR\_Print}, [e1]]\!]^{\perp}], \perp \rangle & \\
A \vdash \langle \texttt{tick}, [\,] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [[\![\texttt{RGR\_Tick}, [\,]]\!]^{\perp}], \perp \rangle & \\
A \vdash \langle \texttt{tock}, [\,] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [[\![\texttt{RGR\_Tock}, [\,]]\!]^{\perp}], \perp \rangle & \\
A \vdash \langle \texttt{debug-state}, [\,] \rangle & \xrightarrow{\mathrm{p}} & \left\langle A, [[\![\texttt{RGR\_DebugKill}, [\,]]\!]^{\perp}], \perp \right\rangle & \\
A \vdash \langle \texttt{forceGC}, [\,] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [[\![\texttt{RGR\_ForceGC}, [\,]]\!]^{\perp}], \perp \rangle &
\end{array}
$$

Figure B.3: Semantics of Dynamix primitives for primitive value creation and debugging

$$
\begin{array}{lcl}
A \vdash \langle \texttt{ineg}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \left\langle A, [\,], [\![\texttt{RGR\_INeg}, [e1]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{iadd}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IAdd}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{imul}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IMul}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{idiv}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IDiv}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{imod}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IMod}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{isub}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_ISub}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{ieq}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \left\langle A, [\,], [\![\texttt{RGR\_IEq}, [e1, e2]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{ilt}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_ILt}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{igt}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IGt}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{ior}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IOr}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{ixor}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IXor}, [e1, e2]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{iand}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IAnd}, [e1, e2]]\!]^{\perp} \rangle \\
 & & \\
A \vdash \langle \texttt{req}, [e1, e2] \rangle & \xrightarrow{\mathrm{p}} & \left\langle A, [\,], [\![\texttt{RGR\_REq}, [e1, e2]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{is-null}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsNull}, [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{is-int}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsInt}, [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{is-frame}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsFrame}, [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{is-code}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsCode}, [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{is-CF}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsCF}, [e1]]\!]^{\perp} \rangle \\
A \vdash \langle \texttt{is-continuation}, [e1] \rangle & \xrightarrow{\mathrm{p}} & \langle A, [\,], [\![\texttt{RGR\_IsCont}, [e1]]\!]^{\perp} \rangle
\end{array}
$$

Figure B.4: Semantics of Dynamix primitives for integer operations and data comparison

$$
\begin{array}{lcl}
A \vdash \langle \texttt{pop}, [\,] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_CFRGet}, [\,]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{curC}, [lbl] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_ContCur}, [lbl]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{newC}, [e1, e2] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_ContNew}, [e1, e2]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{curCF}, [\,] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_CFThis}, [\,]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{newCF}, [e1] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_CFNew}, [e1]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{getC}, [e1, e2] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_ContGet}, [e1, e2]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{setC}, [e1, e2, e3] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [[\![\texttt{RGR\_ContSet}, [e1, e2, e3]]\!]^{\perp}], \perp \right\rangle \\
A \vdash \langle \texttt{unpackC}, [e1] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_ContUnpack}, [e1]]\!]^{\perp} \right\rangle \\
A \vdash \langle \texttt{unpackCF}, [e1] \rangle & \xrightarrow{\mathfrak{p}} & \left\langle A, [\,], [\![\texttt{RGR\_CFUnpack}, [e1]]\!]^{\perp} \right\rangle
\end{array}
$$

Figure B.5: Semantics of Dynamix primitives for control frame operations

Appendix C

Full Dynamix Specification of Scheme

```
module dynamic-semantics

signature
  init-size = 1
  link-labels
    P
  continuations
    ret
  namespaces
    Var
    Line

rules
  eval(Program(exps)) =
    scope <- new(int(0));
    link(scope, [], &P);
    mkcur(scope);
          map[eval-top](exps);
          exitWith(int(0))

  // For some reason SPT breaks when evaluating a Program AST Node
  eval(Test(exps)) =
          eval(Program(exps))

  resolveVar(name) = resolve(name, "Var")

rules
  // In order to follow the Nabl2 specification for Scheme
  // we need to evaluate the expressions used in top-expressions
  // in a new scope.
  // This is not needed from execution point of view.

  eval-top(v@Define(name, exp)); k =
    line-f <- get(cur(), resolve-scope("__NEXT__"@v, "Line"));
    set(line-f, [0], k);

    val <- eval-exp(exp);
    path = resolveVar(name);
```

```
39      scope <- new(int(1));
40      link(scope, [], &P);
41      mkcur(scope);
42      set(scope, path, val);
43
44      jump(get(line-f, [0]))
45
46    eval-top(v@Redefine(name, exp)); k =
47      line-f <- get(cur(), resolve-scope("__NEXT__"@v, "Line"));
48      set(line-f, [0], k);
49
50      v1 <- eval-exp(exp);
51      path = resolveVar(name);
52      set(cur(), path, v1);
53
54      jump(get(line-f, [0]))
55
56    eval-top(exp); k =
57      line-f <- get(cur(), resolve-scope("__NEXT__"@exp, "Line"));
58      set(line-f, [0], k);
59      print(eval-exp(exp));
60
61      jump(get(line-f, [0]))
62
63  rules
64    eval-binop(Add(), l, r) = iadd(l, r)
65    eval-binop(Sub(), l, r) = isub(l, r)
66    eval-binop(Lt(), l, r)  = ilt(l, r)
67    eval-binop(Gt(), l, r)  = igt(l, r)
68    eval-binop(Or(), l, r)  = ior(l, r)
69    eval-binop(And(), l, r) = iand(l, r)
70    eval-binop(Eq(), l, r)  = ieq(l, r)
71
72    eval-binop(Conz(), l, r) =
73      list <- new(int(2));
74      set(list, [0], l);
75      set(list, [1], r);
76      return(list)
77
78    eval-binop(SetCar(), list, val) =
79      v1 <- get(list, [0]);
80      set(list, [0], val);
81      return(v1)
82
83    eval-binop(SetCdr(), list, val) =
84      v1 <- get(list, [1]);
85      set(list, [1], val);
86      return(v1)
87
88  rules
89    eval-unop(Car(), val) =
90      return(get(val, [0]))
```

```
91    eval-unop(Cdr(), val) =
92      return(get(val, [1]))
93
94  rules
95    eval-exp(Num(v))      = return(int(v))
96    eval-exp(EmptyList())= return(new(int(0)))
97    eval-exp(Bool("#t")) = return(int(1))
98    eval-exp(Bool("#f")) = return(int(0))
99    eval-exp(Ref(name))  = return(get(cur(), resolveVar(name)))
100
101   eval-exp(Set(Ref(name), exp)) =
102     v1 <- eval-exp(exp);
103     path = resolveVar(name);
104     v2 <- get(cur(), path);
105     set(cur(), path, v1);
106     return(v2)
107
108
109   eval-exp(BinOp(Gte(), l, r))  =
110     out <- eval-exp(
111       BinOp(Or(),
112         BinOp(Gt(), l, r),
113         BinOp(Eq(), l, r)
114       ));
115     return(out)
116
117   eval-exp(BinOp(Lte(), l, r))  =
118     out <- eval-exp(
119       BinOp(Or(),
120         BinOp(Lt(), l, r),
121         BinOp(Eq(), l, r)
122       ));
123     return(out)
124
125   eval-exp(BinOp(op, left, right)) =
126     v1 <- eval-exp(left);
127     v2 <- eval-exp(right);
128     return(eval-binop(op, v1, v2))
129
130    eval-exp(UnOp(op, exp)) =
131     v1 <- eval-exp(exp);
132     return(eval-unop(op, v1))
133
134   eval-exp(IfElse(cond, then, else)); k(v1) =
135     c <- eval-exp(cond);
136     jumpz(c, else_b, then_b);
137
138     !v1;
139     then_b = <
140       ~v1 <- eval-exp(~then);
141       jump(~k)
142     >;
```

```
143     else_b = <
144       ~v1 <- eval-exp(~else);
145       jump(~k)
146     >
147
148   eval-exp(Let(binds, body)) =
149     o_scope <- cur();
150     scope <- new(int(length(binds)));
151     link(scope, [], &P);
152
153     eval-let-binds(binds, scope);
154     mkcur(scope);
155     res <- eval-let-body(body);
156     mkcur(o_scope);
157     return(res)
158
159
160
161   eval-exp(Lambda(args, body)) =
162     map[associate-args](zip-with-index(args));
163     df <- new(int(length(args)));
164     link(df, [], &P);
165
166     b <- <
167       eval-lamb-body(~body)
168     >;
169     clos <- new(int(2));
170     set(clos, [0], df);
171     set(clos, [1], b);
172     return(clos)
173
174   eval-exp(Callcc(func)); k(pop()) =
175     cc <- curC(k);
176
177     clos <- eval-exp(func);
178
179     cf <- newCF(get(clos, [0]));
180     setC(cf, $ret, cc);
181     df <- unpackCF(cf);
182     set(df, [0], cc);
183     callCF(cf, get(clos, [1]))
184
185
186   eval-exp(FunApp([func | args])); k(pop()) =
187     !closure;
188     call_cont = <
189       arg <- eval-first(~args);
190       callC(~closure, arg)
191     >;
192     call_clos = <
193       cf <- newCF(get(~closure, [0]));
194       setC(cf, $ret, curC(~k));
```

```
195       df <- unpackCF(cf);
196       store-arg(zip-with-index(~args), df);
197       callCF(cf, get(~closure, [1]))
198     >;
199     closure <- eval-exp(func);
200     jumpz(is-frame(closure), call_cont, call_clos)
201
202
203 rules
204    eval-first([first|_]) = eval-exp(first)
205
206    associate-args( (idx, name) ) =
207     associate-index(idx, name, "Var")
208
209   store-arg([], _) = nop()
210   store-arg([(idx, exp) | tail], frame) =
211     v1 <- eval-exp(exp);
212     set(frame, [idx], v1);
213     store-arg(tail, frame)
214
215 rules // Let internal rules
216   eval-let-body([exp]) =
217     v1 <- eval-exp(exp);
218     return(v1)
219
220   eval-let-body([exp | tail]) =
221     v1 <- eval-exp(exp);
222     return(eval-let-body(tail))
223
224   eval-lamb-body([exp]) =
225     v1 <- eval-exp(exp);
226     rc <- getC(curCF(), $ret);
227     callC(rc, v1)
228
229   eval-lamb-body([exp | tail]) =
230     v1 <- eval-exp(exp);
231     eval-lamb-body(tail)
232
233   eval-let-binds([], scope) = nop()
234
235   eval-let-binds([Bind(name, val) | tail], scope) =
236     v1 <- eval-exp(val);
237     set(scope, resolveVar(name), v1);
238     eval-let-binds(tail, scope)
239
```

Figure C.1: Full Dynamix specification of the Scheme implementation from section 7.1.

Appendix D

Full Dynamix Specification of Tiger

```
module dynamic-semantics

signature
  link-labels
    P
    I
  continuations
    ret
    ex
    next
    yield

  namespaces
    Var
    Field
    Loop

  init-size = 15

rules
  eval(m@Mod(exp)) =
    compile-builtins(m);
    eval-instr(exp, m);
    exitOK()

  eval-instr(exp:UNIT(), _) =
          v1 <- eval-exp(exp)

  eval-instr(exp:RECORD(_), _) =
          rec <- eval-exp(exp);
          printc(int(36));
          printc(int(82));
          printc(int(101));
          printc(int(99));
          printc(int(111));
          printc(int(114));
          printc(int(100))

```

```
39    eval-instr(exp:ARRAY(_, _), _) =
40            rec <- eval-exp(exp);
41            printc(int(36));
42            printc(int(65));
43            printc(int(114));
44            printc(int(114));
45            printc(int(97));
46            printc(int(121))
47
48    eval-instr(exp:INT(), m) =
49      v1 <- eval-exp(Call("printi"@m, [exp]))
50
51    eval-instr(exp:STRING(), m) =
52      v1 <- eval-exp(Call("print"@m, [exp]))
53
54  rules
55    compile-builtins(m) =
56      compile-chr(m);
57      compile-print(m);
58      compile-print-int(m);
59      compile-timing(m)
60
61    compile-timing(m) =
62      tick_b = <
63              tick();
64              res <- null();
65              func-return(res, $ret)
66      >;
67      set(cur(), resolve("timeGo"@m, "Var"), tick_b);
68
69      tock_b = <
70              tock();
71              res <- null();
72              func-return(res, $ret)
73      >;
74      set(cur(), resolve("timeStop"@m, "Var"), tock_b)
75
76    compile-chr(m) =
77      chr_b = <
78              val <- get(cur(), [0]);
79              string <- new(int(1));
80              set(string, [0], val);
81              func-return(string, $ret)
82      >;
83      set(cur(), resolve("chr"@m, "Var"), chr_b)
84
85    compile-print(m) =
86      print_b = <
87        dnx_print <- dnx-print-string($ret);
88        cf <- newCF(cur());
89        setC(cf, $ret, curC(~ret));
90        callCF(cf, dnx_print)
```

```
    >;
    ret = <
      func-return(null(), $ret)
    >;
    set(cur(), resolve("print"@m, "Var"), print_b)

  compile-print-int(m) =
    to_string = <
      itos <- dnx-int-to-string($ret);
      df <- new(int(1));
      set(df, [0], get(cur(), [0]));
      cf <- newCF(df);
      setC(cf, $ret, curC(~print_string));
      callCF(cf, itos)
    >;

    print_string = <
      print_s <- dnx-print-string($ret);
      string <- pop();
      df <- new(int(1));
      set(df, [0], string);
      cf <- newCF(df);
      setC(cf, $ret, curC(~ret));
      callCF(cf, print_s)
    >;

    ret = <
      func-return(null(), $ret)
    >;
    set(cur(), resolve("printi"@m, "Var"), to_string)

rules
  eval-exp(Array(_, size, init_val)) =
    s <- eval-exp(size);
    arr <- create-default-array[eval-exp](s, init_val);
    return(arr)

  eval-exp(Subscript(a, i)) =
    array <- eval-exp(a);
    index <- eval-exp(i);
    return(array-get-index(array, index))

  eval-exp(Assign(Subscript(array, index), value)) =
    idx <- eval-exp(index);
    arr <- eval-exp(array);
    val <- eval-exp(value);
    array-set-index(arr, idx, val);
    return(null())


rules
  eval-exp(Plus(left, right)) =
```

```
143       v1 <- eval-exp(left);
144       v2 <- eval-exp(right);
145       return(iadd(v1, v2))
146
147     eval-exp(Gt(e1, e2)) =
148       v1 <- eval-exp(e1);
149       v2 <- eval-exp(e2);
150       return(igt(v1, v2))
151
152     eval-exp(Lt(e1, e2)) =
153       v1 <- eval-exp(e1);
154       v2 <- eval-exp(e2);
155       return(ilt(v1, v2))
156
157     eval-exp(Geq(e1, e2)) =
158       v1 <- eval-exp(e1);
159       v2 <- eval-exp(e2);
160       return(ior(igt(v1, v2), ieq(v1, v2)))
161
162     eval-exp(Leq(e1, e2)) =
163       v1 <- eval-exp(e1);
164       v2 <- eval-exp(e2);
165       return(ior(ilt(v1, v2), ieq(v1, v2)))
166
167     eval-exp(Minus(e1, e2)) =
168       v1 <- eval-exp(e1);
169       v2 <- eval-exp(e2);
170       return(isub(v1, v2))
171
172     eval-exp(Eq(e1, e2:INT())) =
173       v1 <- eval-exp(e1);
174       v2 <- eval-exp(e2);
175       return(ieq(v1, v2))
176
177     eval-exp(Neq(e1, e2:INT())) =
178       v1 <- eval-exp(e1);
179       v2 <- eval-exp(e2);
180       return(isub(int(1), ieq(v1, v2)))
181
182     eval-exp(Eq(e1, e2)) =
183       v1 <- eval-exp(e1);
184       v2 <- eval-exp(e2);
185       return(req(v1, v2))
186
187     eval-exp(Neq(e1, e2)) =
188       v1 <- eval-exp(e1);
189       v2 <- eval-exp(e2);
190       return(isub(int(1), req(v1, v2)))
191
192     eval-exp(Times(e1, e2)) =
193       v1 <- eval-exp(e1);
194       v2 <- eval-exp(e2);
```

```
195       return(imul(v1, v2))
196
197     eval-exp(Divide(e1, e2)); k(idiv(v1, v2)) =
198       !v1;
199       throw_b = <
200         callC(getC(curCF(), $ex), ~v1)
201       >;
202
203       v1 <- eval-exp(e1);
204       v2 <- eval-exp(e2);
205       jumpz(ieq(v2, int(0)), k, throw_b)
206
207     eval-exp(And(e1, e2)) =
208       res <- eval-exp(
209               If(e1,
210                       If(e2, Int(1), Int(0)),
211                       Int(0)
212               )
213       );
214       return(res)
215
216   rules
217     eval-exp(Call(name, args)); k(get-returned-value()) =
218       call-function[eval-exp]((name, "Var"), args, $ret, k)
219
220   rules
221     eval-exp(Int(v)) = return(int(v))
222
223     eval-exp(Uminus(exp)) = return(ineg(eval-exp(exp)))
224
225     eval-exp(Var(name)) = return(variable-get(name, "Var"))
226
227     eval-exp(Seq(list)) =
228       return(eval-seq(list))
229
230     eval-exp(Assign(Var(var), exp)) =
231       v1 <- eval-exp(exp);
232       variable-assign(var, "Var", v1);
233       return(null())
234
235   rules // Sequence internal rules
236     eval-seq([]) =
237       return(null())
238
239     eval-seq([h]) =
240       v1 <- eval-exp(h);
241       return(v1)
242
243     eval-seq([h | t]) =
244       v1 <- eval-exp(h);
245       return(eval-seq(t))
246
```

```
247   eval-seq([h | t]) =
248     v1 <- eval-exp(h);
249     return(eval-seq(t))
250 rules // Try-catch
251   eval-exp(TryCatch(try_exp, Var(var), catch_exp)); k(null()) =
252     try_b = <
253       v <- eval-exp(~try_exp);
254       callC(getC(curCF(), $next))
255     >;
256     catch_b = <
257       val <- pop();
258       set(cur(), resolve(~var, "Var"), val);
259       v <- eval-exp(~catch_exp);
260       callC(getC(curCF(), $next))
261     >;
262
263     try_df   <- new(int(0));
264     catch_df <- new(int(1));
265
266     link(try_df  , cur(), &P);
267     link(catch_df, cur(), &P);
268
269     try_cf   <- newCF(try_df);
270     catch_cf <- newCF(catch_df);
271
272     next <- curC(k);
273
274     setC(try_cf  , $ex  , newC(catch_cf, catch_b));
275     setC(try_cf  , $next, next);
276
277     setC(catch_cf, $next, next);
278
279     callCF(try_cf, try_b)
280
281   eval-exp(Throw(exp)); k(null()) =
282     v <- eval-exp(exp);
283     callC(getC(curCF(), $ex), v)
284
285 rules // For in
286   eval-exp(ForIn(Var(var), gen_exp, body)); k(null()) =
287     var_path = resolve(var, "Var");
288     exec_body_init = <
289       gen <- get(cur(), resolve(~var, "Var"));
290       gen_cf <- unpackC(gen);
291       setC(gen_cf, $yield, curC(~exec_body));
292       callC(gen)
293     >;
294     exec_body = <
295       gen <- pop();
296       val <- pop();
297       set(cur(), resolve(~var, "Var"), val);
298       v <- eval-exp(~body);
```

```
299       callC(gen)
300     >;
301     gen <- eval-exp(gen_exp);
302     for_s <- new(int(1));
303     link(for_s, cur(), &P);
304
305     set(for_s, var_path, gen);
306     setC(unpackC(gen), $ret, exit_c);
307
308     exit_c <- curC(k);
309     for_cf <- newCF(for_s);
310     setC(for_cf, $ret, exit_c);
311     callCF(for_cf, exec_body_init)
312
313   eval-exp(Yield(exp)); k(null()) =
314     val <- eval-exp(exp);
315     callC(getC(curCF(), $yield), val, curC(k))
316
317 rules
318   eval-exp(IfThen(cond, then)); k(null()) =
319    then_b = <
320       v1 <- eval-exp(~then);
321       jump(~k)
322     >;
323
324     c <- eval-exp(cond);
325     jumpz(c, k, then_b)
326
327   eval-exp(If(cond, then, else)); k(v1) =
328     !v1;
329
330     c <- eval-exp(cond);
331     jumpz(c, else_b, then_b);
332
333     then_b = <
334       ~v1 <- eval-exp(~then);
335       jump(~k)
336     >;
337     else_b = <
338       ~v1 <- eval-exp(~else);
339       jump(~k)
340     >
341
342   eval-exp(For(Var(idx_var), init, cond, body)); k(null()) =
343     idx_path = resolve(idx_var, "Var");
344     !f_max;
345     f_init = <
346       ~f_max <- pop();
347       jump(~f_check)
348     >;
349     f_body = <
350       v1 <- eval-exp(~body);
```

```
351       // increment loop index
352       set(cur(), ~idx_path, iadd(get(cur(), ~idx_path), int(1)));
353       jump(~f_check)
354     >;
355     f_end = <
356       callC(getC(curCF(), $break))
357     >;
358     f_check = <
359       cur <- get(cur(), ~idx_path);
360       jumpz(igt(cur, ~f_max), ~f_body, ~f_end)
361     >;
362
363     vi <- eval-exp(init);
364     max <- eval-exp(cond);
365     scope <- new(int(1));
366     link(scope, [], &P);
367     set(scope, idx_path, vi);
368
369     loop_cf <- newCF(scope);
370         setC(loop_cf, $break, curC(k));
371
372     callC(newC(loop_cf, f_init), max)
373
374   eval-exp(While(cond, body)); k(null()) =
375     w_cond = <
376       c <- eval-exp(~cond);
377       jumpz(c, ~w_end, ~w_body)
378     >;
379     w_body = <
380       scope <- new(int(0));
381       link(scope, [], &P);
382
383       mkcur(scope);
384       v1 <- eval-exp(~body);
385
386       mkcur(get(cur(), [&P]));
387       jump(~w_cond)
388     >;
389     w_end = <
390       callC(getC(curCF(), $break))
391     >;
392
393     w_cf <- newCF(cur());
394         setC(w_cf, $break, curC(k));
395     callCF(w_cf, w_cond)
396
397   eval-exp(Break()); k(null()) =
398     callC(getC(curCF(), $break))
399
400 rules
401   eval-exp(Let(binds, body)) =
402     old_s <- cur();
```

```
403       eval-let-binds(binds);
404       res <- eval-let-body(body);
405       mkcur(old_s);
406       return(res)
407
408   // Let internal rules
409   rules // Let body
410     eval-let-body([exp]) =
411       v1 <- eval-exp(exp);
412       return(v1)
413
414     eval-let-body([exp | tail]) =
415       v1 <- eval-exp(exp);
416       return(eval-let-body(tail))
417
418   rules // Let bindings
419     eval-let-binds([]) =
420       nop()
421
422     eval-let-binds([FunDecs(funcs) | tail]) =
423       scope-new(int(length(funcs)));
424       eval-let-binds(funcs);
425       eval-let-binds(tail)
426
427     eval-let-binds([GenDec(name, args, _, body) | tail]) =
428       q = zip-with-index(args);
429       map[associate-args](q);
430       gen_init = <
431         func-return(curC(~gen_body), $ret)
432       >;
433       gen_body = <
434         v1 <-eval-exp(~body);
435         func-void-return($ret)
436       >;
437       variable-assign(name, "Var", gen_init);
438       eval-let-binds(tail)
439
440
441     eval-let-binds([FunDec(name, args, _, body) | tail]) =
442       eval-let-binds([ProcDec(name, args, body) | tail])
443
444     eval-let-binds([ProcDec(name, args, body) | tail]) =
445       q = zip-with-index(args);
446       map[associate-args](q);
447       fun_body = <
448         v1 <-eval-exp(~body);
449         func-return(v1, $ret)
450       >;
451       variable-assign(name, "Var", fun_body);
452       eval-let-binds(tail)
453
454     associate-args( (idx, FArg(name, _)) ) =
```

```
455     associate-index(idx, name, "Var")
456
457   eval-let-binds([VarDec(name, _, val) | tail]) =
458     eval-let-binds([VarDecNoType(name, val) | tail])
459
460   eval-let-binds([VarDecNoType(name, val) | tail]) =
461     v1 <- eval-exp(val);
462     scope-new(int(1));
463     variable-assign(name, "Var", v1);
464     eval-let-binds(tail)
465
466   eval-let-binds([TypeDecs(decs) | tail]) =
467     scope-new(int(0));
468     eval-let-binds(tail)
469
470 rules
471   eval-exp(Record(_, vals)) =
472     return(create-record-named[eval-exp, unpack-field-decls](vals, "Field"))
473
474   eval-exp(FieldVar(record, name)) =
475     rec <- eval-exp(record);
476     return(record-get-field(rec, name, "Field"))
477
478   eval-exp(NilExp()) = return(null())
479
480   eval-exp(Assign(FieldVar(f, name), exp)) =
481     v1 <- eval-exp(exp);
482     rec <- eval-exp(f);
483     record-set-field(rec, name, "Field", v1);
484     return(null())
485
486 rules
487   unpack-field-decls(InitField(name, exp)) = (name, exp)
488
489 rules
490   eval-exp(String(str)) =
491     string <- string-to-fvm-string(str);
492     return(string)
```

Figure D.1: Full Dynamix specification of the Tiger implementation from section 7.2

Appendix E

Dynamix Library Rules

```
module dnx-lib

rules        // General utility
  map-term[x]([]) = []
  map-term[x]([h|t]) =
    h' = x(h);
    t' = map-term[x](t);
    [h' | t']

  map[x]([]) =
    nop()

  map[x]([h|t]) =
    x(h);
    map[x](t)

  zip-with-index(list) =
    zip--with--index(list, 0)

  zip--with--index([], idx) = []
  zip--with--index([h | t], idx) =
    tail = zip--with--index(t, idx + 1);
    [(idx, h) | tail]

  frame-store[x]([], _) = nop()
  frame-store[x]( [(idx, exp) | t ], frame ) =
    val <- x(exp);
    set(frame, [idx], val);
    frame-store[x](t, frame)

  record-store[x]([], _, _) = nop()
  record-store[x]( [(idx, exp) | t ], frame, ns ) =
    val <- x(exp);
    set(frame, [idx], val);
    record-store[x](t, frame, ns)

  record-store-named[x]([], _, _) = nop()
  record-store-named[x]( [(name, exp) | t ], frame, ns ) =
```

```
39      val <- x(exp);
40      set(frame, resolve(name, ns), val);
41      record-store-named[x](t, frame, ns)
42
43  rules // Strings
44    string-to-fvm-string(str) =
45      chars = explode-string(str);
46      string <- new(int(length(chars)));
47      frame-store[eval-char](zip-with-index(chars), string);
48      return(string)
49
50    eval-char(char) =
51      return(int(char))
52
53  rules // Variables
54    variable-get(name, ns) =
55      return(get(cur(), resolve(name, ns)))
56
57    variable-assign(name, ns, value) =
58      set(cur(), resolve(name, ns), value)
59
60  rules // Exitcodes
61    exitOK() =
62      rcf <- getC(curCF(), $ret);
63      callC(rcf, int(0))
64
65    exitWith(code) =
66      rcf <- getC(curCF(), $ret);
67      callC(rcf, code)
68
69  rules  // Arrays
70    array-set-index(array, index, value) =
71      return(set(array, index, value))
72
73    array-get-index(array, index) =
74      return(get(array, index))
75
76    create-default-array[x](size, default_val) =
77      array <- new(size);
78      init--array[x](array, size, default_val);
79      return(array)
80
81    // The size of an array is not constant, so we cannot solve this using meta-arithmetic
82    init--array[x](array, size, exp); k =
83      idx <- int(0);
84      jump(init);
85
86      !idx;
87      init = <
88        jumpz(ieq(~idx, ~size), ~loop, ~k)
89      >;
90      loop = <
```

```
      val <- ~x(~exp);
      set(~array, ~idx, val);
      ~idx <- iadd(~idx, int(1));
      jump(~init)
    >

rules // Functions
  call-function[x]((name, ns), args, ret_c, ret_lbl) =
    callFrame <- new(int(length(args)));
    frame-store[x](zip-with-index(args), callFrame);
    link(callFrame, get(cur(), resolve-scope(name, ns)), &P);
    block <- get(cur(), resolve(name, ns));
    callCF <- newCF(callFrame);
    setC(callCF, ret_c, curC(ret_lbl));
    callCF(callCF, block)

  func-return(val, cont) =
    rcf <- getC(curCF(), cont);
    callC(rcf, val)

  func-void-return(cont) =
    rcf <- getC(curCF(), cont);
    callC(rcf)

  get-returned-value() =
    return(pop())

rules // Scoping
  scope-new(size) =
    scope <- new(size);
    link(scope, [], &P);
    mkcur(scope)

rules // Records
  create-record[x, y](values, field_ns) =
    rec <- new(int(length(values)));
    record-store[x](zip-with-index(map-term[y](values)), rec, field_ns);
    return(rec)

  create-record-named[x, y](values, field_ns) =
    rec <- new(int(length(values)));
    rec_i <- new(int(0));
    link(rec_i, rec, &I);
    record-store-named[x](map-term[y](values), rec_i, field_ns);
    return(rec)


  record-get-field(rec, name, ns) =
    s <- new(int(0));
    link(s, rec, &I);
    val <- get(s, resolve(name, ns));
    return(val)
```

```
143
144   record-set-field(rec, name, ns, val) =
145     s <- new(int(0));
146     link(s, rec, &I);
147     set(s, resolve(name, ns), val)
148
149 rules
150   dnx-print-string(lbl) =
151     !print_idx;
152     !print_max;
153     !string;
154
155     print_b = <
156             ~string <- get(cur(), [0]);
157             ~print_max <- size(~string);
158             ~print_idx <- int(0);
159             jump(~print_init)
160     >;
161     print_init = <
162         cond <- ilt(~print_idx, ~print_max);
163         jumpz(cond, ~print_end, ~print_body)
164     >;
165     print_body = <
166         val <- get(~string, ~print_idx);
167             printc(val);
168             ~print_idx <- iadd(int(1), ~print_idx);
169                 jump(~print_init)
170     >;
171     print_end = <
172             func-void-return(~lbl)
173     >;
174     return(print_b)
175
176   dnx-int-to-string(lbl) =
177         !idx;
178     int-to-string = <
179       scope <- new(int(3));
180           set(scope, [0], get(cur(), [0]));
181           link(scope, cur(), &P);
182           mkcur(scope);
183
184           df <- new(int(1));
185           set(df, [0], get(cur(), [0]));
186           cf <- newCF(df);
187           setC(cf, ~lbl, curC(~int-to-string2));
188           callCF(cf, ~calc-length)
189         >;
190
191         int-to-string2 = <
192           r0 <- pop();
193           set(cur(), [1], r0);
194           jumpz(ilt(get(cur(), [0]), int(0)), ~pos-int-to-string, ~neg-int-to-string)
```

```
    >;

    calc-length = <
      ~idx <- int(0);
      jump(~calc-length-comp)
    >;

    calc-length-comp = <
      ~idx <- iadd(~idx, int(1));
      set(cur(), [0], idiv(get(cur(), [0]), int(10)));
    jumpz(get(cur(), [0]), ~calc-length-end, ~calc-length-comp)
    >;

    calc-length-end = <
      ret <- getC(curCF(), ~lbl);
      callC(ret, ~idx)
    >;

    neg-int-to-string = <
      set(cur(), [0], ineg(get(cur(), [&P, 0])));
      string <- new(iadd(get(cur(), [1]), int(1)));
      set(string, [0], int(45));

      set(cur(), [2], size(string));
      set(cur(), [1], string);
      jump(~int-to-digits)
    >;

    pos-int-to-string = <
      set(cur(), [0], get(cur(), [&P, 0]));
      string <- new(get(cur(), [1]));
      set(cur(), [2], size(string));
      set(cur(), [1], string);
      jump(~int-to-digits)
    >;

    int-to-digits = <
      r0 <- isub(get(cur(), [2]), int(1));
      set(cur(), [2], r0);
      set(get(cur(), [1]), r0, iadd(int(48), imod(get(cur(), [0]), int(10))));
      set(cur(), [0], idiv(get(cur(), [0]), int(10)));
      jumpz(get(cur(), [0]), ~return, ~int-to-digits)
    >;
    return = <
      res <- get(cur(), [1]);
      func-return(res, ~lbl)
    >;
    return(int-to-string)

```

Figure E.1: Dynamix specification of library functions used in Appendix C and D.