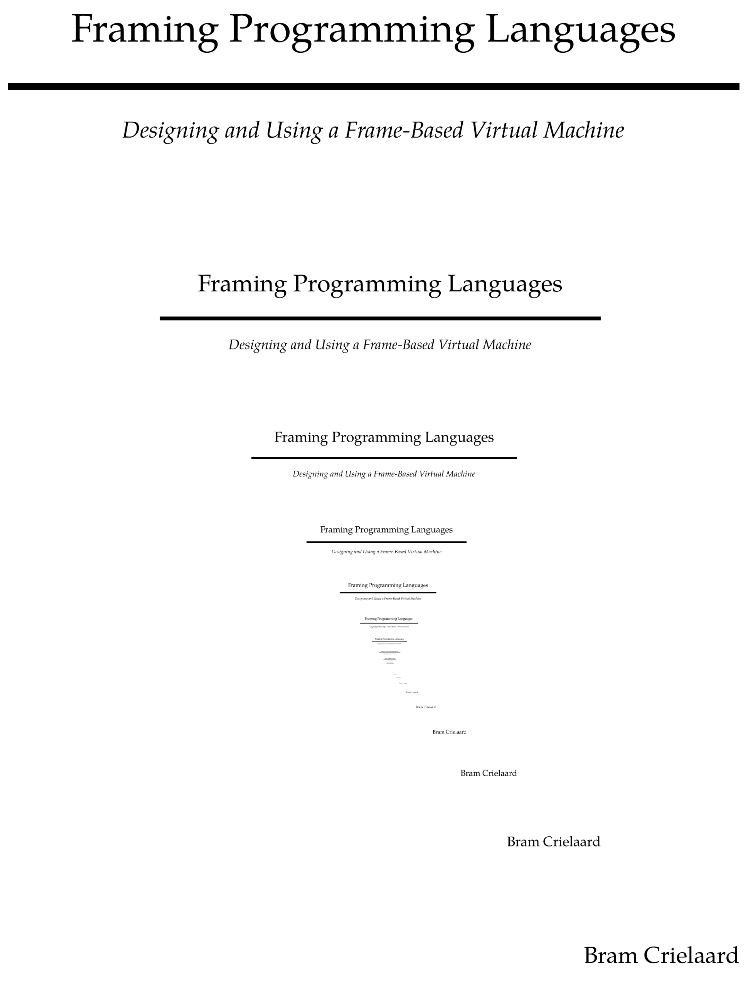
Framing Programming Languages

Designing and Using a Frame-Based Virtual Machine



Framing Programming Languages

Designing and Using a Frame-Based Virtual Machine

Bram Crielaard

Bram Crielaard

Framing Programming Languages

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bram Crielaard
born in Apeldoorn, the Netherlands

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

Cover picture: Cover picture

Source code available on GitHub: `https://github.com/Geweldig/FrameVM`

Framing Programming Languages

Author:	Bram Crielaard
Student id:	4371755
Email:	`framevm@crielaard.co.uk`

Abstract

This thesis introduces the FrameVM virtual machine and the Framed language. This language gives developers a target to compile to which concisely follows the *scopes-as-frames* model. This model allows language developers to derive the memory model based on the scope graphs. The core building blocks of Framed are frames, which contain all data including code. To demonstrate the viability of this model this paper also introduces a compiler from Scheme to Framed, focussing on complex control structures such as *call-with-current-continuation* and closures. As a result we aim to show that Framed is usable as a target language for compiling, even though it does not have a stack nor registers.

Preface

Just like everyone else who comes to the point where they are writing their preface there are a lot of people who I would like to thank. Without their support this thesis would not have ended up the way it has, if it had ever even been finished. In a perfect world, without the current corona-crisis we unfortunately still find ourselves in, this thesis would have reached the quality we were all expecting of it. Yet, here we are.

First of all I would like to thank Eelco Visser, Casper Bach Poulsen, and Andrew Tolmach for their help in guiding this research. I would especially like to thank Eelco for sticking through the drafts of this thesis, even though the quality was not always as high as neither he nor I would have liked.

I would like to thank my girlfriend Marit for her moral support throughout this thesis. I am certain that without her I would not have been able to finish this thesis. She has been a massive help, and her support can not be understated.

Furthermore, I would like to thank my family, Marit her family, Chiel, Luka, Taico, Job, and Jeroen. They will probably know what I want to thank them for, and if not I would like to request they cross out their respective acknowledgement accordingly.

I would also like to thank the entirety of the Programming Languages group, especially Jeff. Without their help when needed I would have hit a wall much earlier in the process, which I fortunately escaped thanks to them.

Last, and most certainly not least, I would like to thank Kevin and Rens. Without their support and input this thesis would not have ended up the way it has. They have provided an innumerable amount of feedback, and have always been available to help me through hard times.

"You may ask yourself — Well... how did I get here?" – *David Byrne*

Bram Crielaard
Tilburg, the Netherlands
April 18, 2021

Framing Programming Languages
Designing and Using a Frame-Based Virtual Machine

Bram Crielaard

Programming Languages Group, Delft University of Technology, Delft, The Netherlands
framevm@crielaard.co.uk
https://pl.ewi.tudelft.nl/

Abstract. This thesis introduces the FrameVM virtual machine and the Framed language. This language gives developers a target to compile to which concisely follows the *scopes-as-frames* model. This model allows language developers to derive the memory model based on the scope graphs. The core building blocks of Framed are frames, which contain all data including code. To demonstrate the viability of this model this paper also introduces a compiler from Scheme to Framed, focussing on complex control structures such as *call-with-current-continuation* and closures. As a result we aim to show that Framed is usable as a target language for compiling, even though it does not have a stack nor registers.

Keywords: Virtual Machines · Scopes-as-frames · Memory Management · Scheme · Control Flow

1 Introduction

Choosing the right programming language for a project is a non-trivial task. Not every language tailors for every type of work, giving us the expression that Python is "the second best language for everything" as you can perform any task with it, with relative ease.

It is desirable to use a language which is the best suited for the project at hand. This can take the form of a domain-specific language (DSL), a programming language which specialises a specific domain, such as the one in the project. We might even have separate parts of a project where we would benefit from different programming languages.

Unfortunately, it is currently not possible to create a DSL for every relevant domain. To create a DSL there are numerous parts we need to implement, such as the type checker and the parser. The Spoofax language workbench[16] is a workbench which strives to provide tools to create both of these parts, among others. It provides SDF3[23] to define the syntax of the language, Statix[4] to define the static semantics, and Stratego[27] to perform program transformations, and DynSem[25] to make writing interpreters easier.

Even though the Spoofax language workbench provides DynSem, we can only use it to write an interpreter for our language. Furthermore, DynSem falls flat on one major detail: We still need to manually design the memory model of our language.

The introduction of the *scopes-as-frames*[21] by Poulsen et al. allows us to reuse our Statix specification to automate our memory model. The *scopes-as-frames* model makes use of scope graphs[3,20] to determine where data should reside in memory. Scope graphs describe the scope of names, and as a result variables, of a program. This maps to the memory model needed by the program, in the form of a frame graph which contains the memory needed to store the variables.

For example, if we look at Figure 1 we can see the scope graph for a Scheme program. Scope 0 contains the declarations of the two lambdas `foo` and `bar`. Scope 1, the scope belonging to the lambda `bar`, in turn contains the declaration of variable `x`. We can create a graph of frames which contain the data needed to store the corresponding variables. If we reach a program state where a scope is no longer accessible we can garbage collect the frame storing the variable.

```
(letrec
  (
    (bar (lambda () (foo 41)))
    (foo (lambda (x) (+ 1 x))))
  (bar))
```

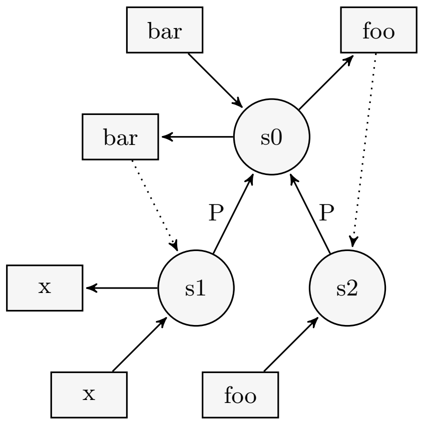


Fig. 1. An example program written in Scheme and the corresponding scope graph

Vergu et al. introduced an extension to DynSem which makes use of this scopes-as-frames paradigm[26]. Even though this brings the speed of interpretation within an order of magnitude of hand written interpreters, it still does not allow for us to easily write a compiler instead of an interpreter.

As a continuation of this Bruin introduced Dynamix[8], which started out as a new target for the DynSem language which would allow for compilation instead of interpretation. However, Bruin introduced Dynamix as a new language as he deemed DynSem did not have good mechanisms for writing down control flow.

The Dynamix language Bruin introduced makes use of a new language called Roger as its compilation target. Roger uses continuations to model control flow such as `call/cc`. Roger encapsulates the scopes-as-frames paradigm as its main language feature, allowing developers to store data in *data frames* and control flow constructions in *control frames*.

We believe this distinction is also one of the main problems with the Roger language, as it is no longer clear to developers in what frame type they should store data. Roger also suffers from a lack of abstraction, forcing the language developer to manually perform complex operations on continuations. An example where we believe Framed drastically improves the readability of compiled code is the "Hello World!" example. We show the example program and the compiled Roger code in Figure 2. We see that we manually need to handle the continuation needed to exit the program, whereas in the Framed example in Figure 3 we do not need to take this into account. This becomes even more complicated when we try to model complex control flow such as exceptions. Bruin himself mentions this as well: "*Writing a program that showcases exception handing [sic] mechanics is one of the most mind-bending programs when the Frame VM is not completely understood yet.*". The final downside is the fact that Roger is a register machine, resulting in memory allocations which are not part of the frame graph.

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

```
MAIN:
        printc(iload(72))
        printc(iload(101))
        printc(iload(108))
        printc(iload(108))
        printc(iload(111))
        printc(iload(32))
        printc(iload(87))
        printc(iload(111))
        printc(iload(114))
        printc(iload(108))
        printc(iload(100))
        printc(iload(33))

        callC(getC(curCF(), $ret), iload(0))
```

Fig. 2. A "Hello World!" program, and the corresponding Roger code[8]
.

To resolve all the mentioned problems with Roger we introduce a new virtual machine called the FrameVM. The goals of this machine are as follows:

1. Allow for a direct encoding of the *scopes-as-frames* paradigm.
2. Do not have any data, including code, outside of frames. This negates the need for the registers used by Roger, while also removing the need for the two distinct frame types.
3. Provide a minimal and complete instruction set. This resolves the problem of us having to manually perform complex operations on continuations.

The FrameVM provides only four possible native data types, namely *Integers*, *Strings*, *Frames*, and *Code Pointers*. There is no distinction between code and data in this machine, as they all live in the same frames. Frames themselves consist of a number of named slots.

The FrameVM does not allow us to store any data outside of frames. This means we need to split operations which create intermediate values, such as `9 + 10 + 23`, and explicitly store their intermediates in a frame. The first of these two operations performs the first addition and stores it in a slot, while the second addition uses the result stored by the previous operation. By using this intermediate slot we will be able to know the number of slots we will need while compiling the program, as there is no unaccounted data in our memory model.

The last goal was to provide a minimal language which we can use to express a wide set of control operations. As a result, we introduce the Framed language which runs on the FrameVM. This language only provides four instructions.

Two research questions guided the design of this language, and in turn this research. The first of these is *whether we could create a usable language without either a stack or registers* based on the *scopes-as-frames* paradigm. The second question is *whether we can model complex control flow operations using such a language*, such as `call/cc` and exceptions.

This paper provides the following contributions:

– We introduce Framed, a new frame-based language (section 2). We also provide an interpreter of the language (section 3), as well as a formal definition (section 4).
– In order to demonstrate the usefulness of this language, we provide a compiler from Scheme to Framed (section 5). Our compiler handles complex control flow operations such as `call-cc` as well as exceptions. We describe the internals of the compiler in the form of compilation schemata.

2 Framed

We use Framed to interact with the FrameVM. This language allows us to perform branching instructions, and assigning values to slots. Framed differs from traditional programming languages as it does not differentiate between data and code.

In this section we will discuss how we can make use of the language and show its syntax in subsection 2.1. Furthermore, to demonstrate the use-cases of Framed we will provide a list of example programs in subsection 2.2.

2.1 Using The Language

We write a Framed program as a frame containing slots, as we show in Figure 3. These slots contain all the code needed to evaluate the program, as well as all data needed by the program. In order for us to write an executable program, we need to populate at least one slot in this frame, namely the `Main` slot, with a frame. We then need to store a code pointer in the `start` slot of this frame. The code in this `start` slot is the code where we will begin execution of the program. In the previously mentioned example we see the two frames, as well as the aforementioned code pointer. Something we have not previously mentioned is what we can actually store in this code pointer. A code pointer consists of a list of instructions. These instructions allow us to evaluate expressions and assign the result to slots, as well as perform (conditional) jumps and write to the output. In this example we show a *base expression*, namely the string "Hello World".

```
frame: [
  Main := frame: [
    start := code {
      show "Hello World!";
    },
  ],
]
```

Fig. 3. A "Hello World" program written in Framed

The Framed language consists of a basic syntax, as we can see in Figure 4.

Here we can see that the instructions take base expressions as their arguments. These base expressions can either be an integer, a string, a code pointer, or a path. When we evaluate a path, we are not interested in the actual path itself, but in the value stored at the location which the path resolves to. There are four values on the FrameVM, namely integers, strings, code pointers, and frames.

r	::=	f	program
b	::=	z	base expression
	|	s	
	|	c	
	|	p	
c	::=	`code` $\{i^+\}$	code pointers
i	::=		instruction
	|	p `:=` e`;`	
	|	`ifeq` b b b`;`	
	|	`jump` b b`;`	
	|	`show` b`;`	

f	::=	`frame: [`d^*`]`	frame
d	::=	x `:=` e`,`	slot
e	::=	b	expression
	|	f	
	|	`!`b	
	|	$b \oplus b$	
p	::=	`self`	path
	|	u	
	|	p`.`x	
u	::=	`^`	path up
	::=	u`.`u	

$z \in \mathbb{Z}$ $s \in$ String $x \in$ Identifier $\oplus \in$ {`+`, `-`, `/`, `*`, `<`, `#`, `==`, `&&`, `||`}

Fig. 4. Syntax specification of Framed

We designed the language in this manner as a compilation target which concisely follows the *scopes-as-frames* paradigm, without having to manually generate the frame graph or a structure to represent it. We believe that writing programs as a nested collection of frames provides a good compromise on readability and expressiveness. A downside of this representation is the fact that it is impossible to write a program where a frame refers to two different frames. However, we believe this trade-off drastically improves the usability of the language, as reasoning about paths becomes much more straightforward than in a language were we could refer to multiple frames.

Paths are one of the main building blocks of the Framed language, we should focus on how we can write them down. There are two main types of paths, namely paths which are relative to the current frame, and paths which are relative to the current `self`. When a program executes, it creates an empty frame, accessible via the `self` keyword[1]. We can use this frame to store any intermediate data we create, meaning we can treat this frame as we would a stack in traditional programming languages. This frame will also become necessary when we discuss the branching instructions.

The other types of paths are relative to the frames in the program. We can access these frames using `^`, which refers to the encapsulating frame in the source program relative to the current code pointer. The interpreter resolves all paths which contain a `^` before running the program. This means that if we change the frame which encapsulates the code pointer during execution, we might try to access paths which no longer exist.

[1] This language was originally inspired by the programming language SELF[24]. However, the keyword is the only real trace of this inspiration which still remains.

We can find an example of where we use both paths relative to self as well as paths relative to the current encapsulating frame in Figure 5. In this example we see the first of the two branching instructions, namely unconditional jumps, on line 5. We also see the assignment instruction on line 4. An assignment stores a value at a specific path, in this case the slot `value` in the current `self`. If a slot does not already exist when writing to it, it is automatically created. We then perform a jump instruction, which takes two arguments. The first of this is the path that we want to jump to, which in this case is the `branch` slot. This path should resolve to a code pointer, as we would otherwise not be able to execute it. The second argument is the frame which should become the new `self` after we have performed the jump. This is useful in the case of function calls, as we want to change the current evaluation context when jumping to a block of code. We show how we can use these in our example programs in subsection 2.2.

```
1  frame: [
2    Main := frame: [
3      start := code {
4        self.value := 0;
5        jump ^.branch self;
6      },
7      branch := code {
8        show self.value;
9      },
10   ],
11 ]
```

Fig. 5. A branching program written in Framed

It is worth noting that we also enforce a limitation when writing down paths. It is only possible to define paths in expressions which are part of a code pointer. This means the following program is not valid:

```
frame: [
  x := 42,
  Main := frame: [
    start := code {},
    y := 42 + ^.x,
  ],
]
```

The final instruction to discuss is the `ifeq` instruction, which we can use to perform a conditional jump. If we look at Figure 6 we can see that the `ifeq` instruction takes three arguments, namely a base expression and two paths. We evaluate the base expression to a value, and if that value is `0` we perform the jump. The two paths work in the exact same manner as the ones for the unconditional `jump` instruction.

```
frame: [
  Main := frame: [
    start := code {
      ifeq 0 ^.branch self;
      show "not branched";
    },
    branch := code {
      show "branched";
    },
  ],
]
```

Fig. 6. A conditionally branching program written in Framed

If we do not perform the jump we fall through to the next instruction in the current code pointer. In the case of the example, we show the string "branched". If we were to change the `0` to a `1` we would instead show the string "not branched".

2.2 Example Programs

To give a more complete overview of how the Framed language works we will show a number of Scheme programs as well as their Framed counterpart, showing how we can encode language constructs.

Functions We will start with an example of compiling a lambda, without invoking it. We can see a Scheme expression which does this here:

```
(lambda (x) (+ x x))
```

Compiling this program gives us the following output:

```
frame: [
  Main := frame: [
    start := code {
      self.ref := frame: [
        function := ^.^.lambda.start,
        parent := self,
      ];
      show self.ref;
    },
  ],
  lambda := frame: [
    // the lambda body
    start := code {
      self.x := self.arg1;
      self.intermediate := self.x + self.x;
      self.caller.callee_result := self.intermediate;
      jump self.return self.caller;
    },
  ],
]
```

We store the body of the lambda in a new frame, named `lambda` in this example. The body loads the argument from the `arg1` slot of the `self` frame and stores it in the named slot `x`. After this it performs the operation in the body of the lambda, `(+ x x)`, and stores this value in the `callee_result` of the `caller` frame. After this it jumps to the `return` slot and passes the `caller` frame as the new same frame.

We store a reference to this lambda in a slot in the current frame, named `ref` in this example. This frame contains two slots, namely the `function` slot which contains the reference to the code, and the `parent` slot which contains a reference to the lexical parent of the lambda.

What might not be immediately obvious is why we show `self.ref` at the end of the program. Scheme evaluates all programs to a value, and prints that value at the end of the execution. As such our compiler also prints the compiled lambda, even though this is in itself not a sensible value to show.

The names of these slots are important, as they are also used by the compiler which we will introduce in section 5. As such we will enumerate them again, and explain their function within a Framed program[2].

A list of these names is as follows:

– `arg1`
  The slot which contains the argument of a function call. Set when invoking a function, and recalled in the body of a function.
– `caller`
  The slot which contains the `self` frame of the caller of the function. We set the current `self` frame to the frame contained in this slot when exiting the function.

[2] Do keep in mind that these names are only a convention, and not an inherit part of Framed. As such it is not required to make use of them.

– `callee_result`
  The slot which will contain the result of a function call. This slot is part of the `caller` frame, which means that `self.callee_result` will always be available after returning from a function.
– `return`
  The slot which contains a reference to the code pointer which we will return to when exiting a function.
– `function`
  The slot which contains a reference to the code pointer which contains the start of a function.
– `parent`
  The slot which contains the lexical parent of a function.

Now that we know how we can compile a function, we will want to evaluate it. As such, let us call the function we previously defined with the number `21` as input:

```
((lambda (x) (+ x x)) 21)
```

Compiling this expression gives us the following Framed program:

```
frame: [
  Main := frame: [
    start := code {
      self.ref := frame: [
        function := ^.^.lambda.start,
        parent := self,
      ];
      // set up the environment needed in the lambda body
      self := frame: [
        return := ^.end_pointer,
        caller := self,
        arg1 := 21,
        parent := self.ref.parent,
      ];
      // invoke the lambda
      jump self.caller.ref.function self;
    },
    end_pointer := code {
      self.res := self.callee_result;
      show self.res;
    },
  ],
  lambda := frame: [
    // the lambda body
    start := code {
      self.x := self.arg1;
      self.intermediate := self.x + self.x;
      self.caller.callee_result := self.intermediate;
      jump self.return self.caller;
    },
  ],
]
```

We invoke a function by creating a new `self` frame. This frame consists of four slots:

- `return`
  We set this slot to the code pointer located in the `end_pointer` slot. This makes sure we can return to the correct code after the execution of the lambda has finished.
- `caller`
  We set this slot to the current `self` frame. We do this, as the current `self` is the caller of the function. This ensures we can return control to the correct frame after executing the lambda.
- `arg1`
  We set this slot to the argument we pass to the lambda, which in this case is the integer `21`.
- `parent`
  We set this slot to `self.ref.parent`, which is the frame which represents the parent scope of the lambda.

We then invoke the lambda with the newly created `self` frame. We do this by jumping to the `start` code pointer of the lambda. As we have already set up our environment when creating the new `self` frame, we return control to the correct frame and in the correct pointer once we have finished executing the lambda.

Exceptions An extension to this construct is the encoding of exceptions. In Scheme we can raise any value as an exception. We start by looking at an expression which only raises an exception with the value `42`, without handling it:

```
(raise 42)
```

Compiling this expression gives us the following Framed program:

```
frame: [
  Main := frame: [
    start := code {
      // set the initial exception handler
      self.exc_handler := frame: [
        handler := frame: [
          function := ^.^.default_handler.start,
          parent := "INVALID",
        ],
        caller := "INVALID",
        return := "INVALID",
        call_exc_frame := "INVALID",
      ];
      // set up the environment needed in the exception handler
      self := frame: [
        return := self.exc_handler.return,
        parent := self.exc_handler.handler.parent,
        caller := self.exc_handler.caller,
        exc_handler := self.exc_handler.call_exc_frame,
        exc_intermediate := self.exc_handler.handler.function,
        arg1 := 42,
      ];
      // raise the exception
      jump self.exc_intermediate self;
    },
  ],
  default_handler := frame: [
  // the default exception handler
    start := code {
      self.res := "uncaught exception: " # self.arg1;
      show self.res;
    },
  ],
]
```

In this example we see that exceptions are an extension of functions. In order for a single raise expression not to crash we set an initial exception handler. This handler consists of four slots:

- `handler`
  We set this slot to the function that we will use for our exception handler. As such it in itself is a frame consisting of the two slots which make up functions.
- `caller`
  We set this slot to the caller of the current exception handler, which we need in the case of nested exceptions. As we are currently setting the initial exception handler there is no valid value for this slot, so we set it to `"INVALID"`.
- `return`
  We set this slot to the return address of the current exception handler, which is also needed in the case of nested exceptions. Just like `caller` before, there is no valid value at this point in the compilation. As a result, we also set it to `"INVALID"`.

- `call_exc_frame`
  We set this slot to the exception handler of our `caller`. As there currently is no caller, we set this to `"INVALID"`.

The reason we need to set all of these slots, even though they are all invalid, is because we do expect them to exist. We can see this when we raise the exception. Raising an exception works similarly to calling a function, but with different slots in the frame. The frame consists of the following slots:

- `return`
  The return code pointer of the exception handler.
- `parent`
  The parent of the current exception handler.
- `caller`
  The caller of the current exception handler.
- `exc_handler`
  The exception handler under which we evaluate our current exception handler.
- `exc_intermediate`
  We use this slot to store an intermediate reference to the current exception handler. We need this as we directly set our current `self` frame to the frame we create here. If we do not set this intermediate we would lose access to the current exception handler, as we lose access to the `self` frame.
- `arg1`
  The value we want to raise, 42 in the case of this example.

Now that we know how we can raise an exception, we will want to catch it. We can see a Scheme program which does this here:

```scheme
(with-handlers (((lambda (x) #t) (lambda (x) x))) (raise 42))
```

This exception handler consists of three parts:

1. The matching function, `(lambda (x) #t)`
2. The function which we will call to handle the exception, `(lambda (x) x)`
3. The expression we are evaluating under the handler, `(raise 42)`

Scheme uses the matching function to see if we should use the current handler, or if we should traverse the stack to find a different one[3]. We invoke the second function with the item we are raising, which is `42` in this case.

Compiling this program gives us the following output:

[3] We force this function to always return true, and as a result we do not actually compile it.

```
frame: [
  Main := frame: [
    start := code {
      // set the initial exception handler
      self.exc_handler := frame: [
        handler := frame: [
          function := ^.^.default_handler.start,
          parent := "INVALID",
        ],
        caller := "INVALID",
        return := "INVALID",
        call_exc_frame := "INVALID",
      ];
      // create the function frame for the new exception handler
      self.intermediate := frame: [
        function := ^.^.handler.start,
        parent := self,
      ];
      // set the current handler to the newly created handler
      self := frame: [
        parent := self,
        exc_handler := frame: [
          return := ^.done,
          caller := self,
          handler := self.intermediate,
          call_exc_frame := self.exc_handler,
        ],
      ];
      // set up the environment needed in the exception handler
      self := frame: [
        return := self.exc_handler.return,
        parent := self.exc_handler.handler.parent,
        caller := self.exc_handler.caller,
        exc_handler := self.exc_handler.call_exc_frame,
        exc_intermediate := self.exc_handler.handler.function,
        arg1 := 42,
      ];
      // raise the exception
      jump self.exc_intermediate self;
    },
    done := code {
      self.res := self.callee_result;
      show self.res;
    },
  ],
  handler := frame: [
    // the with-handlers exception handler
    start := code {
      self.x := self.arg1;
      self.caller.callee_result := self.x;
      jump self.return self.caller;
    },
  ],
  default_handler := frame: [
    // the default exception handler
    start := code {
      self.res := "uncaught exception: " # self.arg1;
      show self.res;
    },
  ],
]
```

In this example we see that before we raise our exception we first create a new exception handler. This exception handler sets all the slots we discussed previously to sensible data, namely the slots of the current default exception handler. After we have created this new handler the program continues as before by creating the environment needed by the raise exception, and then raising the exception.

As we actually have an exception handler now, we enter its body and directly return the value we were raising. As previously mentioned, all Scheme expressions resolve to a single value which is then printed. In our case that is the value we are also raising, as that is the value returned by the exception handler. We can see that the `done` code pointer performs this action.

With the introduction of exceptions and its related slots, we also introduce the following three slots to the convention:

- `exc_handler`
  The slot which contains all the information needed by the exception handler, relative to `self`.
- `handler`
  The slot which contains the function which we will use as the exception handler.
- `call_exc_frame`
  The slot which contains the exception handler which encapsulates the current exception handler.

3 Implementation

We implemented the Framed language, as well as the FrameVM machine used to run it, using the Spoofax language workbench. We will show how we desugar relative paths to absolute paths in subsection 3.1. Furthermore, the FrameVM includes a garbage collector, which we will present in subsection 3.2.

3.1 Path Sugar

As we intend the language to be as minimal as possible there is only one instance of sugar in the syntax, namely the `^` or ”up” for paths. Before a program executes, these paths are statically resolved and rewritten to a top-down access of the slot. In the case of Figure 6, the `show ^.branch` will be rewritten to `show Main.branch`. The reason for this static resolving is the fact that it is possible to change the memory layout of the program during execution. For example we can reference the `Main` frame from two other frames, making the reference ambiguous. This would make resolving the request to go up a frame impossible, as it would suddenly result in two frames being accessible. We show an example of this in Figure 7. In this example both the outer most program frame, as well as the `someSlot` slot in the frame `Other` refer to the same `Main` frame. As a result, the show instruction on line 13 would either refer to the program frame, or the `Other` frame, resulting in an ambiguity.

```
1 frame: [
2   Main := frame: [
3     start := code {
4       ^.^.Other.someSlot := ^.^.Main;
5       show ^.^;
6     },
7   ],
8   Other := frame: [],
9 ]
```

Fig. 7. An example where `^` would become ambiguous if we resolve during runtime

A keen reader will probably have noticed that it is not possible for a programmer to use this top-down access method directly. This is by design, as the path could change if the program is, for example, loaded as a module by a different program. If we were to include a program as a frame in a different program all absolute paths would no longer resolve, as they are now encapsulated by the extra frame. By disallowing absolute paths this problem will never occur.

3.2 Garbage Collection

The FrameVM makes use of a basic mark-and-sweep[6] garbage collection algorithm. For every frame we initialise, we save a reference in the machine. Whenever we initialise a new frame, or assign a new slot, we check if we have not yet reached the maximum amount of memory. We do this by passing all currently known visible frames to the garbage collector, which in our case are the current `self` frame as well as the top level program frame. We can configure the maximum amount of memory by three different metrics, namely the following:

- `MAX_FRAMES`: The total number of frames which may exist at the same time.
- `MAX_SLOTS`: The maximum number of slots which may exist in a single frame.
- `MAX_TOTAL_SLOTS`: The maximum number of slots which may exist over all frames.

Furthermore, we can configure at which percentage of the `MAX_TOTAL_SLOTS` or `MAX_FRAMES` we should invoke the garbage collector. It is not sensible to also check for a limit of `MAX_SLOTS`, as it is not possible to garbage collect slots in a frame. This is the case because whenever a frame is accessible, all slots within it are also accessible.

The garbage collector works by intercepting all calls which deal with memory. Whenever we assign to a slot or initiate a new frame the garbage collector checks if this action follows the conditions. It can check this by keeping track of all frames when we initialise them. When it finds we are at capacity it recursively traverses all visible frames, and marks these as accessible. After we have iterated over all visible frames and marked them we iterate over all frames which exist, deleting all unmarked frames and unmarking all marked frames. All these operations are also logged, so we can see the number of frames the garbage collector has collected when a program has finished running.

4 Formal Semantics of Framed

In this section we will present the formal definition of Framed. We will discuss how we can evaluate base expressions to values in subsection 4.1, and how we can evaluate expressions to values in subsection 4.2. After this we will discuss how we perform instructions in subsection 4.3, and entire programs in subsection 4.4.

A difference between the formal definition of Framed and the Framed discussed in section 2 is the fact that relative paths are not a part of the formal definition, as these would cause ambiguities (subsection 3.1). As such we need to access all slots by traversing the program frame top-down, changing the syntax for paths. We also introduce the syntax for our heap (denoted with the symbol h). We place all frame values (denoted with the symbol σ) we introduce on this heap. If a frame refers to a different frame it makes use of a frame pointer. This frame pointer refers to a location on the heap, which allows us to use frames without copying them.

We provide this altered syntax in Figure 8. The rest of the syntax remains the same.

$$
\begin{array}{rl|rl}
p ::= & \texttt{self} & h ::= & \emptyset \\
\mid & \_ & \mid & x = \sigma;\ h \\
\mid & p.x & \sigma ::= & \emptyset \\
 & & \mid & x = v;\ \sigma
\end{array}
$$

$$x, y \in \text{Identifier} \qquad \sigma = \text{a frame value} \qquad v \in \{c, s, x, z\}$$

Fig. 8. The updated syntax for Framed

To improve readability we do not show the $\emptyset$ as part of our heap or frame value if it is not empty. Also note that frames are not included in the valid list of values. Instead we use the previously introduced frame pointers, which take the form of identifiers. In all judgments Σ and Π are values, that typically are frame pointers to the self frame and program frame respectively.

We define the abstract function `getH(h, x)`, which returns the value located at location x on the heap. For our frame values we define the abstract function `getF(`σ`, x)`, which gets the value stored in slot x of the frame value σ. Both `getH` and `getF` are undefined in case the location or slot we are resolving does not exist.

We also define the abstract function `setH(h, x, `σ`)`, which assigns the frame value σ to the location x on the heap. This function returns an updated heap h'. For our frame values we define the abstract function `setF(`σ`, x, v)`, which assigns the value v to the slot x in frame value σ. This function returns the updated frame value σ'. Both `setH` and `setF` introduce a new location or slot it does not already exist. Otherwise, they overwrite the existing one.

Lastly we introduce the abstract function `initFrame(h)`. This function initialises a frame value on the heap, and returns the tuple $\langle h', x\rangle$, where h' is the updated heap and x is the frame pointer and unique.

4.1 Base Expressions

In order to discuss how we evaluate expressions, we need to know how we evaluate base expressions. We define the evaluation of basic expressions by means of the judgment $\langle b, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v$, where b is a base expression, Σ is the current value representing the `self` frame, Π is the current value representing the program frame, h is the heap, and v is a value.

For integer, string, and code pointer base expressions the values are identical to the base expressions themselves. As such, we can evaluate these without any precondition and we resolve to the base expression itself without altering it.

The way we evaluate path base expressions is more complicated. As we can see in Figure 9 there are two rules which evaluate a path. We use the first of these rules, B-PATH$_1$, to resolve slots in `self` (Σ). We use the second of these rules, B-PATH$_2$, to resolve paths in the program itself (Π).

$$\textsc{B-Int}\quad \frac{}{\langle z, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} z} \qquad \textsc{B-String}\quad \frac{}{\langle s, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} s} \qquad \textsc{B-Code}\quad \frac{}{\langle c, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} c}$$

$$\textsc{B-Path}_1\quad \frac{\langle \Sigma, h, [x_1, \ldots, x_n]\rangle \xrightarrow{Fetch} v \qquad n \geq 0}{\langle \texttt{self.x}_1\texttt{.}\cdots\texttt{.x}_n, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v} \qquad \textsc{B-Path}_2\quad \frac{\langle \Pi, h, [x_1, \ldots, x_n]\rangle \xrightarrow{Fetch} v \qquad n \geq 0}{\langle \texttt{\_.x}_1\texttt{.}\cdots\texttt{.x}_n, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v}$$

$v \in \{c, s, x, z\}$ $\quad$ Σ = value representing the `self` frame
Π = value representing the program frame $\quad$ h = the heap

Fig. 9. Semantics for base expressions

To resolve paths we make use of the fetch rule, which we define in Figure 10. We define the evaluation of fetch by means of the judgment $\langle v, h, xs\rangle \xrightarrow{Fetch} v'$, where v and v' are values, h is the heap, and xs list of identifiers.

Fetch resolves the path relative to the frame pointer we are querying. This means that for a path such as `self.value` we resolve in Σ, while for a path such as `^.branch` we resolve in Π.

4.2 Expressions

Expressions make use of the rules we have introduced in subsection 4.1. They evaluate the base expressions that are part of the expression, and perform the

$$\textsc{Fetch}_1\ \frac{}{\langle v, h, []\rangle \xRightarrow{Fetch} v} \qquad \textsc{Fetch}_2\ \frac{\langle \texttt{getF(getH(h, v), x)}, h, xs\rangle \xRightarrow{Fetch} v'}{\langle v, h, [x|xs]\rangle \xRightarrow{Fetch} v'}$$

$$x \in \text{Identifier} \qquad v \in \{c, s, x, z\}$$

Fig. 10. Semantics for fetch

accompanying operation on the resulting values. We define evaluation of expressions by means of the judgment $\langle e, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h'\rangle$, where e is an expression, Σ is the current value representing the `self` frame, Π is the current value representing the program frame, h is the current heap, v is a value, and h' is the updated heap. The reason for we produce an updated heap is because of the E-Frame rule, which creates new frames on the heap. We present the rules in Figure 11.

In the case of negation (as we can see in E-Not$_1$ and E-Not$_2$) we treat every value other than the integer value `0` as truthy. As such we can negate any base expressions, regardless of what type of value it resolves to. A benefit of this is the fact that we can use any value when performing a branching instruction, as we will show in subsection 4.3.

$$\textsc{E-BinOp}\ \frac{\langle b_1, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v_1 \qquad \langle b_2, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v_2 \qquad v = v_1 \oplus v_2}{\langle b_1 \oplus b_2, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h\rangle}$$

$$\textsc{E-Not}_1\ \frac{\langle b, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} 0}{\langle !b, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle 1, h\rangle} \qquad \textsc{E-Not}_2\ \frac{\langle b, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v \qquad v \neq 0}{\langle !b, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle 0, h\rangle}$$

$$\textsc{E-Frame}\ \frac{\begin{array}{c}\langle h_0, x\rangle = \texttt{initFrame(h)} \qquad \forall 1 \le i \le n.\ \langle e_i, \Sigma, \Pi, h_{\text{i - 1}}\rangle \overset{E}{\Longrightarrow} \langle v_i, h_i\rangle \\ h' = \texttt{setH(}h_n\texttt{, x, (}x_1\texttt{ = }v_1\texttt{; ...; }x_n\texttt{ = }v_n\texttt{;))}\end{array}}{\langle \texttt{frame: [x}_1\texttt{ := e}_1\texttt{, ..., x}_\texttt{n}\texttt{ := e}_\texttt{n}\texttt{]}, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle x, h'\rangle}$$

$$v \in \{c, s, x, z\} \qquad \oplus \in \{\texttt{+, -, /, *, <, \#, ==, \&\&, ||}\}$$

Σ = value representing the `self` frame $\quad$ Π = value representing the program frame
h = the heap

Fig. 11. Semantics for expressions

The E-FRAME rule applies when we introduce a new frame. When creating a frame, we need to evaluate all expressions which we will store in the frame. As such we loop over all expressions, evaluate them to values, and then store these values in a new frame. In the case where we create an empty frame, we will not evaluate any expressions and as a result will not store any of them. This rule adds a new frame to the heap.

4.3 Instructions

All instructions resolve to a tuple, containing an updated list of instructions, an updated `self` value Σ, an updated program value Π, and an updated heap h. We define the evaluation of instructions by means of the judgment $\langle i, cs, \Sigma, \Pi, h\rangle \overset{I}{\Longrightarrow} \langle cs', \Sigma', \Pi', h'\rangle$, where i is an instruction, cs is a list of instructions, Σ is the current value representing the `self` frame, Π is the current value representing the program frame, and h and h' are heaps. We need the list of subsequent instructions cs in order to perform branching operations, as these instructions alter the list.

In the case of the assign rules we make use of the put rule, where we define put in Figure 12. We define the evaluation of put by means of the judgment $\langle v, h, xs, v'\rangle \overset{Put}{\Longrightarrow} h'$, where v and v' are values, h and h' are heaps, and xs is a non-empty list of identifiers.

$$\textsc{Put}_1\quad \frac{}{\langle v, h, [x], v'\rangle \overset{Put}{\Longrightarrow} \texttt{setH(h, v, setF(getH(h, v), x, v'))}}$$

$$\textsc{Put}_2\quad \frac{\langle \texttt{getH(v, x)}, h, xs, v'\rangle \overset{Put}{\Longrightarrow} h'}{\langle v, h, [x|xs], v'\rangle \overset{Put}{\Longrightarrow} h'}$$

$$x \in \text{Identifier} \qquad v \in \{c, s, x, z\}$$

Fig. 12. Semantics for put

We can see in the rules I-IFEQ$_1$ and I-JUMP in Figure 13 that we alter the previously mentioned list cs in the return value. As mentioned in subsection 4.2, we treat all values except for the integer `0` as truthy. As such, we can see in the rules for `ifeq` that we only perform a jump when the condition evaluates to this zero value.

I-Show
$$\frac{}{\langle \texttt{show b}, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, \Sigma, \Pi, h\rangle}$$

I-Jump
$$\frac{\langle b_1, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} cs' \qquad \langle b_2, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} \Sigma'}{\langle \texttt{jump b}_1\ \texttt{b}_2, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs', \Sigma', \Pi, h\rangle}$$

I-IfEq$_1$
$$\frac{\langle b_1, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} 0 \qquad \langle b_2, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} cs' \qquad \langle b_3, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} \Sigma'}{\langle \texttt{ifeq b}_1\ \texttt{b}_2\ \texttt{b}_3, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs', \Sigma', \Pi, h\rangle}$$

I-IfEq$_2$
$$\frac{\langle b_1, \Sigma, \Pi, h\rangle \overset{B}{\Longrightarrow} v \qquad v \neq 0}{\langle \texttt{ifeq b}_1\ \texttt{b}_2\ \texttt{b}_3, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, \Sigma, \Pi, h\rangle}$$

I-Assign$_1$
$$\frac{\langle e, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h'\rangle \qquad \langle \Sigma, h', [x_1, \ldots, x_n], v\rangle \overset{Put}{\Longrightarrow} h'' \qquad n \geq 1}{\langle \texttt{self.x}_1\texttt{.}\cdots\texttt{.x}_n\ \texttt{:= e}, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, \Sigma, \Pi, h''\rangle}$$

I-Assign$_2$
$$\frac{\langle e, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h'\rangle}{\langle \texttt{self := e}, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, v, \Pi, h'\rangle}$$

I-Assign$_3$
$$\frac{\langle e, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h'\rangle \qquad \langle \Pi, h, [x_1, \ldots, x_n], v\rangle \overset{Put}{\Longrightarrow} h'' \qquad n \geq 1}{\langle \texttt{\_.x}_1\texttt{.}\cdots\texttt{.x}_n\ \texttt{:= e}, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, \Sigma, \Pi, h''\rangle}$$

I-Assign$_4$
$$\frac{\langle e, \Sigma, \Pi, h\rangle \overset{E}{\Longrightarrow} \langle v, h'\rangle}{\langle \texttt{\_ := e}, cs, \Sigma, \Pi, h\rangle \overset{I}{\Rightarrow} \langle cs, \Sigma, v, h'\rangle}$$

$v \in \{c, s, x, z\}$ Σ = value representing the `self` frame
Π = value representing the program frame h = the heap

Fig. 13. Semantics for instructions

4.4 Programs

In order for a program to be executable we need to create at least two frame values. The first frame must contain a slot named `Main`, which contains another frame with a slot named `start`. This slot must in turn contain a *code pointer*, which is the code that we execute to start the program. When we invoke a program we apply the rule P-Run, as described in Figure 14. The result of a program is a tuple, containing the final program value Π, final `self` value Σ, and the final heap h. We define the evaluation of programs by means of the judgment $r \Rightarrow \langle \Sigma, \Pi, h\rangle$ where r is the program frame, Σ is the resulting `self` value, Π is the resulting program value, and h is the resulting heap.

The code pointer we resolve and run consists of a list of instructions. We describe the running of these instructions in the rules C-Empty and C-NotEmpty. We apply the former of these rules in case there are no instructions in the code pointer, or if there are no more instructions to execute. We apply the latter rule when there is at least one instruction still available. In this case, we evaluate the first instruction in the list. Because it is possible to jump to a different block of instructions, evaluating an instruction returns the list of instructions which we need to evaluate after the current one has finished. We define the evaluation of code pointers by means of the judgment $cs, \Sigma, \Pi, h \overset{C}{\Longrightarrow} \langle \Sigma', \Pi', h'\rangle$ where cs is a list of instructions, Σ is the `self` value, Π is the program value, and h is the heap.

C-Empty

$$\frac{}{\langle [], \Sigma, \Pi, h\rangle \overset{C}{\Longrightarrow} \langle \Sigma, \Pi, h\rangle}$$

C-NotEmpty

$$\frac{\langle c, cs, \Sigma, \Pi, h\rangle \overset{I}{\Longrightarrow} \langle cs', \Sigma', \Pi', h'\rangle \qquad \langle cs', \Sigma', \Pi', h'\rangle \overset{C}{\Longrightarrow} \langle \Sigma'', \Pi'', h''\rangle}{\langle [c|cs], \Sigma, \Pi, h\rangle \overset{C}{\Longrightarrow} \langle \Sigma'', \Pi'', h''\rangle}$$

P-Run

$$\frac{\begin{array}{c}\langle r, *, *, \emptyset\rangle \overset{E}{\Longrightarrow} \langle \Pi, h\rangle \qquad \langle \Sigma, h'\rangle = \texttt{initFrame(h)} \\ \langle \Pi, h', [\texttt{Main}, \texttt{start}]\rangle \xrightarrow{Fetch} c \qquad \langle c, \Sigma, \Pi, h'\rangle \overset{C}{\Longrightarrow} \langle \Sigma', \Pi', h''\rangle\end{array}}{\langle r\rangle \Rightarrow \langle \Sigma', \Pi', h''\rangle}$$

c = code pointer $\qquad$ Σ = value representing the `self` frame
Π = value representing the program frame $\qquad$ $*$ is any value, as it is never used
r = the program, which takes the form of a frame

Fig. 14. Semantics for entire programs

The P-Run rule needs further explanation, as it uses a magic value `*`. It is impossible for path expressions to be part of the program frame itself, as mentioned in section 2. As a result we will never access Σ nor Π, meaning we can pass any value to the `E-Frame` for either of these. The E-Frame rule in turn evaluates all program slots and creates our heap and program frame pointer Π.

By reusing this rule we do not alter the behaviour of top level frames from frames defined during execution, making the behaviour of Framed more consistent.

5 Compiling Scheme to Framed

In order to test the usability of the Framed language, we developed a compiler for Scheme. We chose Scheme as it contains the three main interesting control flow constructs we wanted to compile: closures (subsection 5.4), `call/cc` (subsection 5.5), and exceptions (subsection 5.6). There are numerous dialects of Scheme, such as MIT-Scheme[15] and Racket[12], which have behavioural differences. Because of this we chose a specific dialect as reference for the compiler, namely Racket.

The compilable subset consists of the following expressions, which we chose to cover a wide range of control operations:

1. Binary Expressions	(`+, *, >, equal?`)
2. Unary Expressions	(`not`)
3. Variables	(`let, letrec, set!`)
4. Exceptions[4]	(`raise, with-handlers`)
5. Conditionals	(`and, or, if`)
6. Continuations	(`call-with-current-continuation`)
7. Procedures[5]	(`lambda`)
8. Literals	(`#t, #f, integers`)
9. Miscellaneous	(`writeln, begin`)

The compiler makes use of an internal language called FrameFlat. This language differs from Framed by allowing programmers to ignore one of its defining features, namely slots. Instead of having to define slots we can define labels, which we use to indicate the boundaries of slots. Because of this we can create a flat list of instructions in a frame, which compile to slots containing code pointers.

The reason for this distinction is the fact that branching logic can generate instructions for multiple code pointers. This in itself is not a major issue, however the problem manifests when we try to compile an expression after a branching expression. This compilation step either needs to be aware of its context in order to store the instructions in the correct code pointer, or it needs to be able to return a list of instructions which we need to append to the aforementioned list. As a result the introduction of the FrameFlat language makes it a lot easier to develop compilers for Framed. We show an example of a FrameFlat program and the corresponding Framed program in Figure 15.

We will give an explanation of how we can compile FrameFlat to Framed in subsection 5.1. After this we will present how we can compile Scheme to FrameFlat, starting with a number of base expressions in subsection 5.2. Once we have explained this, we will present how we compile closures in subsection 5.4. This in turn will more easily allow us to explain how we compile call/cc in subsection 5.5, and how we compile exceptions in subsection 5.6.

[4] Again, the only valid guard allowed for `with-handlers` is the lambda `(lambda (x) #t)`. We believe this is a fair limitation, as we can still emulate the behaviour of different guards in the handler body. As such we believe being able to compile guards was not important for this research.

[5] Lambdas only accept a single argument.

```
  self.value := 0;
  jump ^.branch self;
lbl branch;
  show self.value;
```

⇒

```
frame: [
  Main := frame: [
    start := code {
      self.value := 0;
      jump ^.branch self;
    },
    branch := code {
      show self.value;
    },
  ],
]
```

Fig. 15. A FrameFlat program and the corresponding Framed program

5.1 Compiling FrameFlat

We make use of an intermediate language when compiling Scheme to Framed, the syntax of which we can find in Figure 16. This language does not have code pointers, but instead has labels. We can generate a list of instructions in our compiler without having to explicitly know where code pointers begin and end. However, we are still able to generate frames directly as part of an expression. We need this to, for example, generate a function frame as we describe in section 2.

$$
\begin{array}{lll}
r ::= i^{+} & & \text{program} \\
\\
b ::= z & & \text{base expression} \\
\quad\mid s & & \\
\quad\mid p & & \\
\\
i ::= & & \text{instruction} \\
\quad\mid p \texttt{ := } e\texttt{;} & & \\
\quad\mid \texttt{ifeq } b\ b\ b\texttt{;} & & \\
\quad\mid \texttt{jump } b\ b\texttt{;} & & \\
\quad\mid \texttt{show } b\texttt{;} & & \\
\quad\mid \texttt{lbl } x\texttt{;} & &
\end{array}
\qquad
\begin{array}{lll}
f ::= \texttt{frame: [}d^{*}\texttt{]} & & \text{frame} \\
\\
d ::= x \texttt{ := } e\texttt{,} & & \text{slot} \\
\\
e ::= b & & \text{expression} \\
\quad\mid f & & \\
\quad\mid \texttt{!}b & & \\
\quad\mid b \oplus b & & \\
\\
p ::= \texttt{self} & & \text{path} \\
\quad\mid u & & \\
\quad\mid p\texttt{.}x & & \\
\\
u ::= \texttt{^} & & \text{path up} \\
\quad ::= u\texttt{.}u & &
\end{array}
$$

$z \in \mathbb{Z}$ $\qquad s \in$ String $\qquad x \in$ Identifier $\qquad \oplus \in \{$`+`, `-`, `/`, `*`, `<`, `#`, `==`, `&&`, `||`$\}$

Fig. 16. Syntax specification of FrameFlat

In order to convert this flat representation of the slots to an actual frame we introduce the transformation `fromFlat`. This transformation takes a list of instructions as input, and returns a list of slots as output. We can find the definition of this function in Figure 17.

```
fromFlat([])              ⇒

fromFlat([lbl x; | ii])   ⇒   x := code {
                                  ii
                              },
                      where lbl _; ∉ ii

fromFlat([lbl x; | ii])   ⇒   x := code {
                                  ii'
                              },
                              res
                      where ii' ← takeUntil(?lbl _;) ii
                            res ← fromFlat(dropUntil(?lbl _;) ii)
```

Fig. 17. The `fromFlat` function needed to convert FrameFlat to Framed

In these rules, the `?lbl _` refers to us iterating through the list until we find a label. So in other words we divide the list into parts, starting with labels. We then use the labels as the name for a slot, and we use the rest of the instructions as the body of the slot. As such, compiling an entire program works as follows:

```
[[ e ]]_prog              ⇒   frame: [
                                Main := frame: [
                                  s
                                ],
                                f,
                              ]
                      where [[ e ]] ⇒ (v, i, f)
                                i' ← appendShow(i, v)
                                 s ← fromFlat([lbl start; | i'])
```

In these rules `appendShow` appends a `show v` instruction if `v` is not the string `#<void>`. Invoking a `writeln` expression returns this string, but it will not be automatically printed if it is the resulting value of a program. As such, the compiler should not automatically generate code to print it. We define the judgment for compilation as `[[ e ]] ⇒ (v, i f)`, where `e` is the expression to compile, `v` is the value this expression generates (so either the slot where we can find the result of the expression, or the value itself), `i` is the list of instructions needed to evaluate the expression, and `f` is a list of frames the expression generates. This list of frames is useful in case we compile, for example, a lambda. Being able to move the code belonging to a lambda to a separate frame increases the readability and understandability of the generated Framed program.

This compilation schema also shows us that compiling an expression returns more than just a list of instructions. When compiling an expression, we return the value of that expression, the instructions that expression generates, and the frames that expression generates. The reason why we return all of these, will become clearer in the next subsection.

For examples of how all these schemata work in practice we recommend downloading the compiler introduced in this paper, and going through the example programs provided.

5.2 Compiling expressions

In this section we will present three expressions, namely integers, binary expressions, and `if` expressions. These three cover the main techniques which the compiler makes use of.

Integers We start off by showing how we compile an integer:

```
[[ i ]]                 ⇒  (
                             i,
                             < >,
                             < >
                           )
```

As we can see compiling an integer returns a tuple. This tuple, as mentioned in subsection 5.1, contains the value of the expression, the instructions needed to evaluate the expression, and the frames this expression generates. We see an integer directly returns its value without generating any code, meaning we directly inject this value into further expressions in the program. If we look at the Scheme program `(begin 1 2)` we see that the value `2` is the only value we actually use, as this is the resulting value of the program. As we directly inject the values we will only inject this value into a show expression, meaning there is no reference to the `1` in the compiled program. A downside of this is that decompilation to the exact program becomes impossible, as we have lost the information that there was a sequence in the source program.

Binary Expressions In scheme binary operations take the form of the operator, followed by two expressions. We can see an example of this previously mentioned value injection when compiling these expressions:

```
[[ (⊕ e1 e2) ]]           ⇒   (
                                self.ρ,
                                <
                                  i1
                                  i2
                                  self.ρ := v1 ⊕ v2;
                                >,
                                <
                                  f1
                                  f2
                                >
                              )

                     where [[ e1 ]] ⇒ (v1, i1, f1)
                           [[ e2 ]] ⇒ (v2, i2, f2)
                                 ρ ← fresh
```

In order to perform a binary expression on two values we first need to evaluate the expression which generates them. This means we first perform the instructions to evaluate `e1`, followed by the instructions to evaluate `e2`. We then store the resulting value of the binary operation in an intermediate value, which is the resulting value of the binary expressions. If we perform a binary expression on two integers, both `i1` and `i2` will be empty, as we do not actually generate any instructions. This means we will end up with a single line of code in order to evaluate the binary expression, where we have injected the integers into the expression itself. The following example shows how this happens:

```
[[ (+ 1 2) ]]             ⇒   (
                                self.ρ,
                                <
                                  self.ρ := 1 + 2;
                                >,
                                < >
                              )

                     where [[ 1 ]] ⇒ (1, < >, < >)
                           [[ 2 ]] ⇒ (2, < >, < >)
                                ρ ← fresh
```

Branching Expressions Now that we have presented the compilation of primitive values and base expressions, we can look at how we compile branching expressions. We start by presenting the compilation of `if`. The method with which we compile branching expressions exemplifies the need for the distinction between FrameFlat and Framed. We jump from one code pointer to another, so we need to introduce labels we can jump to. If we instead generated slots directly we would never be able to add more code to an existing slot, or we would have to generate partial slots resulting in invalid Framed code.

In Scheme `if` expressions take the form of the keyword `if`, followed by three expressions. The first of these expressions, `e1`, is the condition. The following expression, `e2`, is the expression that we evaluate in case the condition is true. The last expression, `e3`, is the expression that we evaluate in case the condition is false.

Compiling an `if` expression to FrameFlat is similar to how we would traditionally compile it to a language such as JVM bytecode. We perform the instructions needed to evaluate the condition, and jump if the condition evaluates to false. If it does not, we fall through and evaluate the `true` branch of the if statement. If we had jumped, we would have ended up in a code pointer where we would instead evaluate the `false` branch. The compilation works as follows:

```
[[ (if e1 e2 e3) ]]        ⇒  (
                                self.ρ1,
                                <
                                    i1
                                    ifeq v1 ^.ρ2 self;
                                    i2
                                    self.ρ1 := v2;
                                    jump ^.ρ3 self;
                                  lbl ρ2;
                                    i3
                                    self.ρ1 := v3;
                                    jump ^.ρ3 self;
                                  lbl ρ3;
                                >,
                                <
                                  f1
                                  f2
                                  f3
                                >
                              )

                      where [[ e1 ]] ⇒ (v1, i1, f1)
                            [[ e2 ]] ⇒ (v2, i2, f2)
                            [[ e3 ]] ⇒ (v3, i3, f3)
                                  ρ1 ← fresh
                                  ρ2 ← fresh
                                  ρ3 ← fresh
```

A detail which might stand out is the fact that we jump to relative paths. We do this because there are constructs which introduce new frames. This forces us to use a relative path in order to jump to the correct code pointer. Therefore using paths such as `^.`ρ_2 makes sense.

5.3 Let Bindings

In Scheme we can bind multiple variables in the same let binding. A let binding takes the following form:

```
(let ((x e1)) e2)
```

In this example we bind the result of the expression `e1` to the variable `x`. We then evaluate `e2` in an environment where `x` exists. The reason for the duplicate brackets in the let binding is because we can bind multiple variables in the same binding:

```
(let ((x1 e1) (x2 e2)) e3)
```

A notable limitation to these bindings is the fact that we can not directly refer to a previously bound variable this way. An example of such an invalid expression is as follows:

```
(let ((x1 1) (x2 x1)) x2); <- invalid
```

However, if we bind a variable to lambda which references a different variable this works fine, such as in this example:

```
(let ((x1 e1) (x2 (lambda () x1))) e3)
```

Scheme does provide a variation of the let bindings, `letrec`, where it is possible to refer to previously bound variables directly. However, the evaluation order of these bindings differs from implementation to implementation. As our compiler follows the definition as provided by Racket we evaluate these bindings from left to right. We provide the compilation schema for `letrec` in appendix A.

When we compile a let binding we want to reassign the current `self` in such a way that it contains all the variables defined in the let binding. We first evaluate all bindings, after which we introduce a new self frame which has a slot that points to the old self frame, which we call the parent frame. The new frame will also contain all variables which we are binding. We then evaluate the body of the expression, and store the result of the evaluation in a slot in the parent frame. We do this as the value would otherwise become inaccessible. After we have finished evaluating the body of the let binding, we reset `self` to the parent frame and leave the current scope.

We show this in the following compilation schema:

```
[[ (let ((x1 e1) ... (xn en)) e2) ]]
              ⇒ (
                  self.ρ,
                  <
                    i1
                    ...
                    in
                    self := frame: [
                      parent := self,
                      x1 := v1,
                      ...,
                      xn := vn,
                    ];
                    i2
                    self.parent.ρ := v2;
                    self := self.parent;
                  >,
                  <
                    f1
                    ...
                    fn
                    f2
                  >
                )

                     where [[ e1 ]] ⇒ (v1, i1, f1)
                           ...
                           [[ en ]] ⇒ (vn, in, fn)
                           [[ e2 ]] ⇒ (v2, i2, f2)
                                  ρ ← fresh
```

We pass along the value `self.`ρ to the next compilation step, as the previous `parent` frame is now our `self` frame. An important detail of how we compile let bindings is the fact that `self` is immediately overwritten with a new frame. If we were to first create the frame and store it in an intermediate slot in the `self` we would be unable to let it go out of scope, resulting in the garbage collector being unable to free it.

However, just being able to store variables is not entirely useful as we would also like to be able to retrieve them. Fortunately, this is trivial because we follow the *scope-as-frames* model. When resolving a variable in the scope graph we determine the number of parent edges we need to follow in order to reach the declaration. Afterwards we inject the correct number of `parent` retrievals in the path. It is noteworthy that this is not how we resolve variables if there is a call/cc in the program. We will explain how this differs in subsection 5.5.

5.4 Compiling Closures

Closures in Scheme take the following form:

```
(lambda (x) e)
```

Here `x` is the parameter which refers to the argument we pass to the lambda, and `e` is the body of the lambda. To compile a closure there are two things we need to keep track of: the code pointer which contains the body of the lambda, and in which scope we defined it. We show this in the following schema:

```
[[ (lambda (x) e) ]]      ⇒  (
                               self.$\rho_1$,
                               <
                                   self.$\rho_1$ := frame: [
                                     function := ^.^.$\rho_2$.start,
                                     parent := self,
                                   ];
                               >,
                               <
                                 f
                                 $\rho_2$ := frame: [
                                   s
                                 ],
                               >
                             )

                      where [[ e ]] ⇒ (v, i, f)
                               $\rho_1$ ← fresh
                               $\rho_2$ ← fresh
                                s ← fromFlat(<
                                    lbl start;
                                      self.x := self.arg1;
                                      i
                                      self.caller.callee_result := v;
                                      jump self.return self.caller;
                                     >)
```

Here we see that, for the first time, we introduce a new frame on which we invoke `fromFlat` directly. This frame makes up the closure itself. By convention this frame consists of a slot `start`, which is the slot we jump to when we invoke the lambda. The first thing we do in the body of the closure is copy the value of the argument to the named variable in the body. After this we append the instructions which make up the body of the lambda. In order to be able to return from the closure we store the result of the body in a slot in the caller. By convention this is the `callee_result` slot. After this, we jump to the return code pointer as provided in the frame which we created when invoking the closure, and set the `self` frame to the `self` frame of the caller.

The instructions the compilation generates create a type of ”function frame”. This frame consists of a `function` and a `parent` slot. The `function` slot contains a reference to the start slot of the closure frame we discussed previously. The parent slot contains the current self at the time of compilation. The reason for this parent slot is the fact that we can reference variables outside of the lambda body in the lambda. In order for us to be able to find them in our scope-graph we need access to the parent scope at the time we compile the closure.

Invocation In order to invoke a closure, we first need to retrieve the ”function frame” which defines it. We create a new frame which will be the ”call frame” for the closure. This ”call frame” contains the return address of the closure, the caller to which we should return control, the argument passed to the closure, and the parent scope of the closure.

After we have created this ”call frame” we jump to the code pointer which contains the instructions belonging to the closure. After we have jumped back from the closure, we copy the result value to a temporary slot, potentially wasting a slot. We perform this copy because in the case of recursive functions this slot could be overwritten, but this is not always needed. It is possible to optimise this by performing data flow analysis on the source program. For the purpose of this compiler we have not implemented this.

We show how we compile a closure call in the following schema, where e1 is the closure and e2 is the argument:

```
[[ (e1 e2) ]]            ⇒   (
                               self.ρ1,
                               <
                                   i1
                                   i2
                                   self := frame: [
                                     return := ^.ρ2,
                                     caller := self,
                                     arg1 := v2,
                                     parent := v1.parent,
                                   ];
                                   jump self.caller.v1.function;
                                 lbl ρ2;
                                   self.ρ1 := self.callee_result;
                               >,
                               <
                                 f1
                                 f2
                               >
                             )

                     where [[ e1 ]] ⇒ (v1, i1, f1)
                           [[ e2 ]] ⇒ (v2, i2, f2)
                                 ρ1 ← fresh
                                 ρ2 ← fresh
```

Because of the way we compile closures we can reuse this invocation code when invoking a continuation. As such, all the complexity when compiling a `call-with-current-continuation` lies in the setup needed to create a closure which correctly sets up the environment when invoked. We will show how we actually achieve this in subsection 5.5.

5.5 Compiling call/cc

In Scheme `call-with-current-continuation` is an operator which takes a single expression[6]. This works as follows:

```
(call-with-current-continuation (lambda (x) e))
```

Here the lambda receives the continuation of the `call/cc` as its argument. The body of the lambda can then choose to call this continuation, or ignore it. This leads to interesting control flow behaviour, as in the case of the following example:

```
(+ 1 (call-with-current-continuation (lambda (x) (+ 2 (x 3)))))
```

In this program the lambda receives the continuation `(+ 1 □)`, where the result of its evaluation fills the hole in the binary expression. In the case of our example we call the continuation with the value `3`. That means the expression we evaluate to will be equal to `(+ 1 3)`. This means that by invoking the continuation we can ”break out of” the expression `(+ 2 (x 3))`.

We can also decide to not invoke the continuation:

```
(+ 1 (call-with-current-continuation (lambda (x) (+ 2 3))))
```

In the case of this example we return to the continuation `(+ 1 □)` when we exit the lambda body. This means that the expression we evaluate is equal to `(+ 1 (+ 2 3))`. As we show here we can model complex control flow using just `call/cc`.

As previously mentioned, compiling `call-with-current-continuation` is not too different from compiling a closure, as we show in the following schema:

[6] In the case of our compiler this expression has to be a lambda, as for all other programs we can remove the `call/cc` as the continuation is not used.

```
[[ (call-with-current-continuation e) ]]
              ⇒ (
                  self.ρ₁,
                  <
                      i
                      self := frame: [
                        return := ^.ρ₃,
                        caller := self,
                        arg1 := frame: [
                          function := ^.ρ₂,
                          parent := self,
                        ]
                        parent := v.parent,
                      ];
                      jump self.caller.v.function self;
                    lbl ρ₂;
                      self.parent.callee_result := self.arg1;
                      self := self.parent;
                      jump ^.ρ₃ self;
                    lbl ρ₃;
                      self.ρ₁ := self.callee_result;
                  >,
                  <
                    f
                  >
                )

                    where [[ e ]] ⇒ (v, i, f)
                               ρ₁ ← fresh
                               ρ₂ ← fresh
                               ρ₃ ← fresh
                               isPath(v)
```

In order to invoke the handler of the call/cc we need to pass it a continuation. This continuation contains the same two slots as the ”call frame” we introduced when discussing closures. In this case the `arg1` slot contains the continuation. This continuation is equal to the ”function frame” in the case of closures. The function slot in this frame is a link to a slot containing the next expression after evaluating the call/cc, in the case of our examples this is equal to `(+ 1 □)`. In general this is the code pointer we would jump to after we have evaluated the `call/cc`.

We see that ρ_3 is where we will continue after the `call/cc`, as we set this as the return code pointer for our call frame. ρ_2 is the continuation we pass to the lambda. The need for distinction between these two code pointers exists because we need to correctly set up our environment when we invoke the continuation manually. In the case where we do not invoke the continuation, the lambda we invoked will return control itself. If we invoke the continuation with a value we still need to return control the correct frame. We do this in the ρ_3 slot.

As all data on the FrameVM is mutable it is also possible to model delimited continuations using the current compiler[22]. However, a direct implementation of delimited continuations would be beneficial, but we left this as an exercise to the reader.

A difference from normal compilation which we have not yet discussed, but is important when compiling `call/cc`, is how we retrieve variables. Normally we access variables directly. If we were to do that for programs which make use of `call/cc`, it is possible that they return an incorrect result. This is the case as all variables are mutable, meaning it is possible to overwrite the variable when invoking the continuation. As such, whenever a program contains a call/cc instruction, we store the retrieval of all variables in intermediate slots.

5.6 Compiling Exceptions

Now that we have discussed how closures and continuations work, we can see that exceptions are a straightforward expansion of these two. Before we discuss how we can compile them, we need to discuss a limitation to Scheme which our compiler imposes.

In Scheme, exceptions take the following form:

```
(with-handlers (((lambda (x) #t) e1)) e2)
```

In this example the lambda `(lambda (x) #t)` is a guard. We can use this guard to determine whether the current handler should handle the exception. If the guard returns true we will use the current handler, otherwise we will raise the exception again. We do not allow the guard to return false, which means we can only use the lambda used in the example. We decided on this limitation as we can model the behaviour of a guard in the body of the exception handler, which in our example is the expression `e1`. This expression takes the form of a lambda, which takes the value we raise as an argument. The last expression `e2` is the expression we are evaluating under the handler.

If a `raise` exists in the program we add a slot named `exc_handler` to the `self` frame. This slot contains a frame which stores all information needed to raise an exception. Because of this we need to copy over this exception handler every time we introduce a new `self` frame. As a result the way closures and let bindings work changes when we introduce a raise expression to the source program. The way we actually compile an entire program also differs. If we raise an exception without setting a handler we want the program to still exit "cleanly". As a result, if we have a raise in the program we add the following instruction to the `Main.start` slot:

```
self.exc_handler := frame: [
  handler := frame: [
    function := ^.^.ρ1.start,
    parent := "INVALID",
  ],
  caller := "INVALID",
  return := "INVALID,
  call_exc_frame := "INVALID",
];
```

This instruction sets the initial exception handler. We set a lot of the slots to the value `"INVALID"`. This is because there is no valid value to assign to these slots at this point in time. If these are ever used by any other part of the code the program would crash, which is what we expect to happen, since there is no valid program which accesses these values. However, we still must set these slots, since we copy these values. As such not setting them will not solve the problem of the program exiting cleanly, as we will still be addressing non-existent slots.

A keen reader will also have noticed that we reference a frame here which we have not yet introduced. This is the case because we also introduce a new frame at the top level of the program, which contains the handler code for the exception. This handler makes sure we exit the program cleanly, and print the same error message Racket prints. This frame works as follows:

```
ρ1 := frame: [
  start := code {
    self.ρ2 := "uncaught exception: " # self.arg1;
    show self.ρ2;
  },
],
```

This handler takes the value we are raising, and appends it to the string `uncaught exception:`, after which it prints that string.

Now that we know how the base case works, the way that we compile the `with-handlers` and the `raise` becomes straightforward. First we will show the schema for how to compile a `raise` expression:

```
[[ (raise e) ]]
           ⇒ (
               "INVALID",
               <
                 i
                 self := frame: [
                   return := self.exc_handler.return,
                   parent := self.exc_handler.handler.parent,
                   caller := self.exc_handler.caller,
                   exc_handler := self.exc_handler.call_exc_frame,
                   ρ := self.exc_handler.handler.function,
                   arg1 := v,
                 ];
                 jump self.ρ self;
               >,
               <
                 f
               >
             )

                     where [[ e ]] ⇒ (v, i, f)
                                ρ ← fresh
```

From this schema it becomes clear that raising an exception is, again, the invocation of a closure with some setup. An interesting aspect is the intermediate slot which contains the handler function. The reason we copy this function is because we set the `self` to a new frame. If we did not copy this value the handler would go out of scope before we could call it. Another important detail is the value that we pass as the result of the `raise` expression. This is, as seen previously, the `"INVALID"` string. We can not write a valid program where we use this value, but we must return a value.

Raising an exception changes the exception handler. This becomes necessary in case we have nested handlers, as this would be the handler of the current handler. We show how this works, and how we compile `with-handlers`, in the following schema:

```
[[ (with-handlers (((lambda (x) #t) e1)) e2) ]]
              ⇒ (
                  self.ρ₁,
                  <
                      i1
                      self := frame: [
                        parent := self,
                        exc_handler := frame: [
                          return := ^.ρ₂,
                          caller := self,
                          handler := v1,
                          call_exc_frame := self.exc_handler
                        ]
                      ];
                      i2
                      self.parent.callee_result := v2;
                      jump ^.ρ₂ self.parent;
                    lbl ρ₂;
                      self.ρ₁ := self.callee_result;
                  >,
                  <
                    f1
                    f2
                  >
                )

                     where [[ e1 ]] ⇒ (v1, i1, f1)
                           [[ e2 ]] ⇒ (v2, i2, f2)
                                 ρ₁ ← fresh
                                 ρ₂ ← fresh
                                 isPath(v1)
```

This setup creates a new `self` frame which contains the new exception handler. It then executes the expression `e2` which we evaluate under the handler, after which it will jump to a new block of code. The reason for this is because we need to end up in a join point, as it is not certain if we raise an exception. Because of this we need to be able to guarantee we end up at the same block of code whether we raise an exception or not. As such, we create the ρ_2 label which will be the return point of the exception handler as well as the jump of the aforementioned expression `e2`, which gives us the join point we want.

5.7 Optimisations

There are three optimisations we apply when compiling the code:

– We do not copy variables to intermediate slots if there is no call/cc in the program
– We do not inject handler code if there is no `raise` in the program

– We combine nested `let`/`letrec` where possible

As mentioned in subsection 5.5, whenever we reference a variable in a program which contains a call/cc we copy the variable to an intermediate slot. As it is impossible to end up in a case where this is necessary without a call/cc in the program, we do not do this if there is none in the program.

As seen in subsection 5.6, we inject a lot of instrumentation code when we want to raise an exception. This consists of the `exc_handler` slot, the frame with the values for the default exception handler, as well as the default exception handler itself. As all of these are not necessary when we do not raise an exception, we do not include these when there is no raise in the program. We give an example of the resulting code this generates in Figure 31.

```
(+ (with-handlers (((lambda (x) #t) (lambda (x) 1337))) 10) 32)
```

```
frame: [
  Main := frame: [
    start := code {
      self.s_0 := 10 + 32;
      show self.s_0;
    },
  ],
]
```

Fig. 31. An example of a Racket program which makes use of an exception handler, but will never raise an exception

As we see in this example, the handler is not part of the code. Neither are all the frames needed to keep track of the current exception handler. This cleans up the compiled code a lot, but does make decompiling of the original code impossible, as this exact program is also produced if we were to compile `(+ 10 32)`.

The last optimisation is not specific to this compiler, but it is a useful one. If we have a let/letrec bind which only contains another let binding as its body, we merge them if allowed. This means that instead of introducing two new scopes, which both need a frame, we only introduce a single scope. We show an example of this optimisation in Figure 32. This of course is not always possible, for example in the case the inner let bind refers to the variable declared in the outer let bind. In such a case we do not apply the optimisation, as it would create an invalid program.

```
(let ((x 1)) (let ((y 2)) y))
; gets optimised to
(let ((x 1) (y 2)) y)
```

Fig. 32. An example of how the compiler optimises a let bind

5.8 Testing

In order to test the compiler, we wrote 154 example programs in Scheme. We use a script to run all these programs using the official Racket interpreter and capture the output. We then combine the input programs and the output the Racket interpreter generates to create a Spoofax test suite in SPT, which compiles and runs all the programs using the Scheme to Framed compiler. This gives us a reasonable indication that the compiler we introduce in this paper compiles common programs correctly.

Furthermore, we wrote a separate test suite to test the optimisations described in subsection 5.7. These test that applying the optimisations actually changes the code that we generate. This is helpful, as we otherwise only test the behaviour of the generated code and ignore the generated code itself.

5.9 Known Limitations

There are three known limitations of the current implementation.

Boolean Conversion The first of these is that we convert booleans to integers in the generated Framed code. This means that we can compile certain invalid Scheme programs such as the one seen in Figure 33, even though they are not valid Scheme code. We could solve this by wrapping all values in a frame, containing an integer representing the value and a second value representing the type. However, this would be memory intensive and difficult to work with.

```
(+ 1 #t) ; prints 2 when compiled and ran
```

Fig. 33. An invalid Racket program which we can compile

Incorrect `writeln` Output The second known issue is the fact that we do not always handle the `(writeln e)` instruction correctly. We convert this call to a `show e` instruction in Framed. This is not always correct, as it does not take into account how Scheme prints things such as lambdas. For example, the program shown in Figure 34 crashes when executed. This would be non-trivial to fix, as we would need to write instrumentation code in Framed to handle the conversion of a lambda to the correct string. For the purpose of this research we only implemented the `writeln` instruction for debugging purposes. As such, we deemed it a fair compromise for it to work most but not all of the time.

```
(writeln (lambda (x) x))
```

Fig. 34. A Racket program which crashes when executed

Exceptions Affect Compilation Schemata The last known issue is the fact that exceptions change the way we compile other expressions. As we currently copy the exception handler whenever we introduce a new `self` frame, we need to be aware of exceptions when compiling. This means that the compilation schema for `raise` and `with-handlers` might break if we add more expressions to the compiler. As such, a rewrite of the way we compile exceptions is desirable.

6 Evaluation of Framed

In this paper we introduced a new language, Framed, with the main goal of providing a DSL which allows for a direct encoding of the *scopes-as-frames* paradigm. In this section we will evaluate whether we have achieved this goal, as well as the goal of allowing the compilation of complex control flow.

6.1 Encoding of *scopes-as-frames*

In section 1 we outlined the following two goals for the Framed language, with regards to the *scopes-as-frames* paradigm:

- Allow for a direct encoding of the *scopes-as-frames* paradigm.
- Do not have any data, including code, outside of frames.

In Framed we write all programs as a flat representation of our frame graph. It does not make use of a heap or registers, but instead requires developers to store all data in frames. Furthermore, all code is also stored in the frames, making it clear which parts of source programs produce which frames and slots.

A downside of our encoding of the `scopes-as-frames` paradigm, which we have previously addressed in section 2, is the fact that it is not possible to nest frames under multiple frames. While it is possible to introduce a scope graph where this is necessary, we do not provide a direct encoding of this in Framed. However, we believe the increase in readability we have achieved by this trade-off makes it worthwhile. Furthermore, it is still possible to encode these graphs in Framed, but not directly. At runtime it is possible to build a frame graph where a frame nests under multiple frames, allowing us to still provide a suitable language for these edge cases.

Because of this we believe we have achieved our goal of providing an encoding of the `scopes-as-frames` paradigm without the need to manually create the frame graph, except for the situation previously outlined.

6.2 Encoding Complex Control Flow

In order to demonstrate the ability of Framed to compile complex control flow we introduced a compiler from Scheme to Framed (section 5). This compiler focusses on the compilation of `call/cc` as well as exceptions. We have demonstrated that it is possible to model these operations, even though we explicitly limited the instruction set available in Framed.

The existence of this compiler demonstrates the usability of Framed as a compilation target for real program languages. Furthermore, the compiler demonstrates that a large part of the scope graph is reusable when generating Framed code, making it a well suited compilation target.

Because of this we believe we have achieved our goal of providing a compilation target which allows for the encoding of complex control flow.

7 Related Work

We have introduced a new virtual machine and corresponding language in order to concisely work with the *scopes-as-frames* model. We will discuss the previous work which tried to achieve this by Bruin in subsection 7.1. Furthermore, our work on trying to provide a generalised target language for compilation is not unique. As such, we will discuss the current state of the art of compilation targets in subsection 7.2.

We will also discuss work on compiling Scheme in subsection 7.3, using continuation passing style (CPS) in subsection 7.4, and a possible usage of Framed as a backend language for dynamic semantic specifications in subsection 7.5.

7.1 Frame VM by Bruin

Bruin developed a virtual machine which makes use of *scopes-as-frames*[8]. In this machine there are two types of frames, namely *control frames* and *data frames*. As the names suggest the data frames contain all data needed to evaluate a program such as variables, while the control frames contain information such as the program counter.

This machine has seen usage as part of two research projects aside from the original work by Bruin[9,19]. Unfortunately, there are difficulties when developing for it.

First of all, the distinctions between the aforementioned two types of frames are not explicitly clear. For example, it is possible to store all data in a control-frame, meaning we can write an entire program without using a single data frame.

Second of all, the semantics of the machine are complex. In order to write a valid program numerous intricate operations such as getting and setting continuations need to be manually written by the programmer.

The third pain point of the machine implemented by Bruin is the fact that there is data which exists outside of the frame graph. There are registers available which can store intermediate values such as computation results which can transcend frames, meaning there is data which is not accounted for by the *scopes-as-frames model*. These registers are unfortunately not the only way data can exist outside of the model. All the code which makes up a program also exists outside of the frame graph, making it unclear where it should exist.

We address these issues in our language Framed. All data in our language exists in the same frames, negating the need for the differentiation between control and data frames. We address the complex semantics by providing a machine with fewer instructions and without registers. By doing this we force developers to split complex operations and make their intermediates explicit, making them part of the frame graph.

7.2 Generalised Compilation Targets

There are numerous compilation targets available. In this subsection we will discuss three common known ones, and explain how they differ from the FrameVM introduced in this paper.

LLVM LLVM (Low-Level Virtual Machine)[17] is a compiler framework, which provides a low-level code representation. LLVM is a lot more low level than the FrameVM, and requires direct interaction with the memory we abstract over. It does not specify a runtime model, or object model. LLVM makes use of a stack and virtual registers, where the FrameVM does not make use of either. It aims to be complementary to high level virtual machines such as the JVM we will discuss later.

As a result, LLVM does not fit the purpose of the FrameVM we have introduced in this paper. Where the LLVM allows for a lot more freedom, the FrameVM restricts developers to the `scopes-as-frames` paradigm. One could see this as a downside, but we believe this is instead a benefit of using the FrameVM. The FrameVM forces a pragmatic approach to developing compilers for it.

However, we do believe that LLVM can be a useful target for the FrameVM to compile to. This would provide the best of both worlds: We enforce the *scopes-as-frames* paradigm, but still achieve the speed possible with a low-level language.

WebAssembly Haas et al. introduced WebAssembly[14]in 2017 as a target for binary code, focussing on usage in web browsers. Like the FrameVM, WebAssembly is defined in terms of formal semantics. A WebAssembly binary takes the form of a module, which handles the memory for that binary. WebAssembly makes use of *linear memory*, which takes the shape of a large array of bytes. This memory has a defined length, but can be dynamically extended by using special instructions. This differs from memory on the FrameVM, where memory is implicitly created by the form of the program. This means the developer does not have to take it into account, which they do when using WebAssembly.

WebAssembly also differs from the FrameVM in the way it handles control flow. WebAssembly does not provide a jump instruction, but instead makes use of *structured control flow*. This takes the form of more high level functions, which guarantee safety when branching. A downside of this approach is the fact that complex control flow such as continuations are not natively possible[2]. We can see that this is a current issue, as the Scheme to WebAssembly compiler Schism by Google[13] does not provide call/cc[1]. However, workarounds are available to make it possible on WebAssembly.

As a result, even though there is a lot of merit in WebAssembly we believe it does not fill the void that the FrameVM attempts to fill. FrameVM aims to provide a memory agnostic way to model complex control flow operations, which WebAssembly currently does not provide. Again, just like LLVM, we do believe that WebAssembly could be an interesting target for the FrameVM itself.

JVM The JVM[18] is a virtual machine currently developed by Oracle. Programs on the JVM take the form of class files, similar to programs in the Java language itself. The JVM itself is a stack machine, with access to a constant pool which can be used for fields. It provides abstractions for function calls, and even provides a goto statement if needed.

A downside of the JVM is the fact that modelling frames is cumbersome. It would either have to take the form of a class, in which case it is impossible to introduce a new slot at runtime, or use a generic data-type to contain the frames[7]. This means that a lot of memory is potentially wasted when modelling the *scopes-as-frames* paradigm. While it is possible to write a compiler which follows the `scopes-as-frames` paradigm with the JVM as a target, a lot of boilerplate code is needed. Framed and in turn the FrameVM negate the need for this boilerplate code, making programs easier to read and write.

7.3 Compiling Scheme

There are numerous active research projects related to compiling Scheme. Two noteworthy examples of these are Pycket which introduces JIT compilation[7], and the reimplementation of Racket in Chez Scheme by Flatt et al.[11].

While these compilers focus on speed and coverage of the Scheme language, our compiler focusses on neither of these. We focus on a new way of representing all data needed by Scheme in memory, with speed and coverage being an afterthought.

7.4 Continuations

The main principles behind how our compiler compiles closures and continuations are not novel. Appel has discussed the main ideas behind Continuation Passing Style (CPS) and using this principle in a compiler[5]. Furthermore, Danvy[10] has also written about the usage of CPS conversion for compilation and interpretation.

As CPS is already a major part of Scheme itself it was not needed for this compiler to perform a CPS transformation of the source language. However, the concepts described in these papers still provide valuable information when compiling a program in CPS.

7.5 DynSem and Dynamix

DynSem[25,26] and Dynamix[8] are two languages which are part of the Spoofax Language Workbench[16]. Vergu and Bruin introduced these languages to make it easier for language designers to write interpreters and compilers for their DSLs. These languages both support the *scopes-as-frames* principle. With the introduction of the Framed language these languages could both use Framed as its backend. As such, we believe the language introduced in this paper is a valuable asset to make DSL creation easier.

[7] This is how the current implementation works, as it makes use of HashTables on the JVM.

8 Future Work

As with any research project there are things which were out of the scope of this research. In this section we will discuss six of these items, which we would still like to implement in the future.

8.1 Rewrite Interpreter

We wrote the current implementation of the FrameVM in Stratego[27] using the Spoofax language workbench. A downside of this is the fact that we run on the Java Virtual Machine (JVM). This means we do not have direct access to memory. To still be able to interact with them we modelled them using hashtables. In our future work we would like to rewrite the interpreter in a lower level language. This would allow us to create an actual data object which represents our frames.

Furthermore, this would also make our garbage collector actually useful. In its current form we could remove the entire garbage collector without leaking any data. This is the case because the JVM garbage collector already collects all the frames which go out of scope. With the introduction of our custom garbage collector this no longer works, meaning we have to manually flag frames for collection. This means that the added benefit of the garbage collector is purely theoretical. Reimplementing it in a lower level language would mean we have full control of it, making it an actual useful addition to the interpreter.

An alternative approach would be compiling Framed to either LLVM or WebAssembly, as hinted towards in section 7. Both of these would allow for Framed to be usable in a wider setting, and broaden its appeal.

8.2 Improve Tooling

In order to compile Scheme we have written tooling related to the compilation of the scopes. Even though these make use of a Statix definition, they are still not language agnostic in their current form. As a result, if we were to compile a different language to Framed we would need to reimplement these tools. In the future we would like to rewrite all this tooling to become language agnostic, meaning this issue should no longer arise for future compilers. This would allow us to more easily compile more languages in the future, without having to redo part of our previous efforts.

8.3 Delimited Continuations

As mentioned in section 5 it is theoretically possible to compile delimited continuations using just the control structures we currently have. However, in the future we would like to implement delimited continuations into the Scheme compiler. This would further demonstrate the power of Framed, and would be a useful reference for further control structures.

8.4 Rework Exceptions

In section 5 we mention that we should compile exceptions in a different manner, and we should implement this in the future. The way this should be possible includes two changes, one to the Framed language itself and another to the compiler.

Changes to Framed First of all, the Framed language will need a new expression. This expression will check whether or not a certain slot exists in a frame. We can then use this to conditionally perform instructions based on the condition that we can find a specific slot.

Changes to the Compiler Using the aforementioned expression we can, instead of copying our exception handler, unwind our `caller` stack until we find our handler. This means that the compilation of expressions such as `let` no longer needs to be aware of the fact that there is a `raise` in the program.

Another benefit of this change would be the fact that we no longer needlessly duplicate data, as we will only store the exception handler once instead of duplicating it every time we introduce a new `self` frame.

8.5 Reduce Slot Count

Currently we always create a new slot when we introduce a new intermediate variable. This is, naturally, not technically necessary. We use all intermediate variables (at most) once. This means that we should be able to reuse a lot of the intermediate slots we create. This would lower the memory footprint of our programs, while not bringing on any downsides.

Furthermore, we also create intermediate slots which are never accessed. As a result, we should be able to not generate these. However, actually implementing this would mean we would need to perform data-flow analysis on the code after compilation.

Another reason as to why we have not implemented these is the fact that they should both be part of the Framed language itself, and not of the Scheme compiler. However, this language currently only consists of a syntax definition and an interpreter. Implementing the aforementioned data-flow analysis to resolve these two problems is non-trivial, especially as all the setup to create this is also not yet implemented. As such, we have left this to future work.

8.6 Self-Modifying Code

Even though it is currently possible to modify the contents of slots at runtime, it is not possible to generate new code pointers. This means that it is not possible to specialise a program during execution. In the future we would like to research a way for Framed to be able to alter and generate code pointers during execution. We believe this would be a powerful tool, as code pointers are the last pieces of memory which can not be altered during execution.

9 Conclusion

In this research we set out to create a minimal language based on the *scopes-as-frames* model which gives us the ability to model complex control flow. With the introduction of the new FrameVM and the accompanying Framed language we believe we have achieved our goal of creating such a language. Furthermore, with our introduction of a Scheme to Framed compiler we also believe we have demonstrated the ability of the Framed language to model complex control flow.

As our Framed language does not make use of stacks nor registers we also believe we have answered our first research question, *whether we could create a usable language without either a stack or registers*, and that it is possible to create a functional language with these restrictions.

Furthermore, we believe our Scheme compiler answers our second research question, *whether we can model complex control flow operations using such a language*, and shows us that this is possible in our language.

In conclusion we believe we have achieved our goals outlined in this paper, and have introduced a new programming language which can serve as an important building block for future research.

10 Acknowledgements

We would like to thank Eelco Visser, Casper Bach Poulsen, and Andrew Tolmach for their support and insights during this research. We would also like to thank all people who have supported us through this research, as well as provided reviews of drafts of this work.

References

1. Add call/cc · Issue #75 · google/schism (2019), https://github.com/google/schism/issues/75
2. Continuations · Issue #1252 · WebAssembly/design (2019), https://github.com/WebAssembly/design/issues/1252
3. van Antwerpen, H., Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A constraint language for static semantic analysis based on scope graphs. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 49–60. ACM (2016), http://doi.acm.org/10.1145/2847538.2847543
4. van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proceedings of the ACM on Programming Languages **2**(OOPSLA) (2018), https://doi.org/10.1145/3276484
5. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
6. Appel, A.W.: Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press (2002)

7. Bauman, S., Bolz, C.F., Hirschfeld, R., Kirilichev, V., Pape, T., Siek, J.G., Tobin-Hochstadt, S.: Pycket: a tracing jit for a functional language. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. pp. 22–34. ACM (2015), http://doi.acm.org/10.1145/2784731.2784740
8. Bruin, C.: Dynamix on the Frame VM: Declarative dynamic semantics on a VM using scopes as frames. Master's thesis, Delft University of Technology (2020), https://repository.tudelft.nl/islandora/object/uuid:ddedce14-65ad-4f16-912e-6b0658eaecc0
9. Crielaard, B., Beinema, E.: Compiling Rust to the FrameVM (2019)
10. Danvy, O.: Defunctionalized interpreters for programming languages. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008. pp. 131–142. ACM (2008), http://doi.acm.org/10.1145/1411204.1411206
11. Flatt, M., Derici, C., Dybvig, R.K., Keep, A.W., Massaccesi, G.E., Spall, S., Tobin-Hochstadt, S., Zeppieri, J.: Rebuilding racket on chez scheme (experience report). Proceedings of the ACM on Programming Languages **3**(ICFP) (2019), https://doi.org/10.1145/3341642
12. Flatt, M., Yu, G., Findler, R., Felleisen, M.: Adding delimited and composable control to a production programming environment. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007. pp. 165–176. ACM (2007), http://doi.acm.org/10.1145/1291151.1291178
13. Google LLC: Schism (2018), https://github.com/google/schism
14. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: 0001, A.C., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 185–200. ACM (2017), http://doi.acm.org/10.1145/3062341.3062363
15. Hanson, C., The MIT Scheme Team: MIT/GNU Scheme Reference Manual. Free Software Foundation (2018)
16. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010. pp. 444–463. ACM, Reno/Tahoe, Nevada (2010), https://doi.org/10.1145/1869459.1869497
17. Lattner, C., Adve, V.S.: Llvm: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004), http://csdl.computer.org/comp/proceedings/cgo/2004/2102/00/21020075abs.htm
18. Lindholm, T., Yellin, F., Bracha, G., Buckley, A., Smith, D.: The Java Virtual Machine Specification, Java SE 16 Edition. Oracle (2021)
19. Miljak, L.: Building a Compiler from Prolog to FrameVM (2019)
20. Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A theory of name resolution. In: Vitek, J. (ed.) Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp. 205–231. Springer (2015), http://dx.doi.org/10.1007/978-3-662-46669-8_9

21. Poulsen, C.B., Néron, P., Tolmach, A.P., Visser, E.: Scopes describe frames: A uniform model for memory layout in dynamic semantics. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy. LIPIcs, vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016), http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20
22. Shan, C.: Shift to control. In: In ACM SIGPLAN Scheme Workshop, Snowbird (2004)
23. de Souza Amorim, L.E., Visser, E.: Multi-purpose syntax definition with SDF3. In: de Boer, F.S., Cerone, A. (eds.) Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12310, pp. 1–23. Springer (2020), https://doi.org/10.1007/978-3-030-58768-0_1
24. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: OOPSLA. pp. 227–242 (1987)
25. Vergu, V.A., Néron, P., Visser, E.: Dynsem: A DSL for dynamic semantics specification. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland. LIPIcs, vol. 36, pp. 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015), http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365
26. Vergu, V.A., Tolmach, A.P., Visser, E.: Scopes and frames improve meta-interpreter specialization. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. LIPIcs, vol. 134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2019), https://doi.org/10.4230/LIPIcs.ECOOP.2019.4
27. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In: Lengauer, C., Batory, D.S., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers. Lecture Notes in Computer Science, vol. 3016, pp. 216–238. Springer (2003), https://doi.org/10.1007/978-3-540-25935-0_13

A Compilation Schemata

```
[[ (not e) ]]              ⇒   (
                                 self.ρ,
                                 <
                                   i
                                   self.ρ := !v;
                                 >,
                                 <
                                   f
                                 >
                               )

                       where [[ e ]] ⇒ (v, i, f)
                                  ρ ← fresh
```

```
[[ #t ]]                  ⇒   (
                                1,
                                < >,
                                < >
                              )
```

```
[[ #f ]]                  ⇒   (
                                0,
                                < >,
                                < >
                              )
```

```
[[ (letrec ((x1 e1) ... (xn en)) e2) ]]
             ⇒ (
                  self.ρ,
                  <
                    self := frame: [
                      parent := self,
                    ];
                    i1
                    self.x1 := self.v1;
                    ...
                    in
                    self.xn := self.vn;
                    i2
                    self.parent.ρ := v2;
                    self := self.parent;
                  >,
                  <
                    f1
                    ...
                    fn
                    f2
                  >
               )

                    where [[ e1 ]] ⇒ (v1, i1, f1)
                          ...
                          [[ en ]] ⇒ (vn, in, fn)
                          [[ e2 ]] ⇒ (v2, i2, f2)
                                 ρ ← fresh
```

```
[[ (writeln e) ]]          ⇒   (
                                 "#<void>",
                                 <
                                   i
                                   show v;
                                 >,
                                 <
                                   f
                                 >
                               )

                      where [[ e ]] ⇒ (v, i, f)

[[ (begin e1 e2) ]]        ⇒   (
                                 v2,
                                 <
                                   i1
                                   i2
                                 >,
                                 <
                                   f1
                                   f2
                                 >
                               )

                      where [[ e1 ]] ⇒ (v1, i1, f1)
                            [[ e2 ]] ⇒ (v2, i2, f2)

[[ (and e1 e2) ]]          ⇒   (
                                 self.ρ₁,
                                 <
                                     i1
                                     ifeq v1 ^.ρ₂;
                                     i2
                                     ifeq v2 ^.ρ₂;
                                     self.ρ₃ := 1;
                                   lbl ρ₂;
                                     self.ρ₁ := 0;
                                     jump ^.ρ₃;
                                   lbl ρ₃;
                                 >,
                                 <
                                   f1
                                   f2
                                 >
                               )

                      where [[ e1 ]] ⇒ (v1, i1, f1)
                            [[ e2 ]] ⇒ (v2, i2, f2)
                                  ρ₁ ← fresh
                                  ρ₂ ← fresh
                                  ρ₃ ← fresh
```

```
[[ (or e1 e2) ]]          ⇒   (
                                self.ρ1,
                                <
                                    i1
                                    ifeq v1 ^.ρ2;
                                    self.ρ1 := 1;
                                    jump ^.ρ4;
                                  lbl ρ2;
                                    i2
                                    ifeq v2 ^.ρ3;
                                    self.ρ1 := 1;
                                    jump ^.ρ4;
                                  lbl ρ3;
                                    self.ρ1 := 0;
                                    jump ^.ρ4;
                                  lbl ρ4;
                                >,
                                <
                                  f1
                                  f2
                                >
                              )

                      where [[ e1 ]] ⇒ (v1, i1, f1)
                            [[ e2 ]] ⇒ (v2, i2, f2)
                                  ρ1 ← fresh
                                  ρ2 ← fresh
                                  ρ3 ← fresh
                                  ρ4 ← fresh
```