A Continuation-based Framework for Control Structures in Scheme

Michael Hannema
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.p.r.hannema@student.utwente.nl

ABSTRACT

We develop a framework that simplifies the modelling, implementation and usage of control structures in the Scheme programming language. The framework includes a visual language for modelling control structures. This language can be translated unambiguously to specifications in Scheme, which can in turn be automatically processed to yield the desired control structures. The implementation of the framework is based on first-class continuations and allows the programmer to implement his own control structures without dealing directly with continuations and their perceived complexity.

Keywords

Meta Programmings, Syntactics, Macros, Language Constructs, Control Flow, Flow Graphs, Continuations, Scheme

1. INTRODUCTION

Most programming languages offer a fixed set of control structures that are used to specify control flow; this is an important aspect of the expressivity of the language and influences the programmer's reasoning. The continuation is a control structure that can be used by the programmer to extend a language with new control structures. Adding custom control structures to a language makes it easier to read, write and reason about common programming idioms that have no standard facilities, provided that the implementation details are abstracted away in some fashion. This research aims to develop a visual model of control structures and a way to transform this model into user-friendly control structures in the Scheme programming language.

This paper is structured as follows. The research goals are stated in section 2. Section 3 contains background information that is needed to understand the results of the research, which are covered in sections 4 to 8.

2. RESEARCH GOALS

The general aim of this research is to design a framework that can aid a programmer in implementing control structures. When designing a control structure, it would be convenient to first specify the desired control flow in a visual language; specifically a control flow graph tailored for this task. A model like this, when unambiguous, could then be used as a design specification. Thus, the first goal of this research is to develop a graph form suitable for describing control structures.

After modelling the control structure, it should be implemented in a programming language. The continuation is a powerful control mechanism that is suitable to implement control structures. However, source code that uses continuations directly can be very hard to comprehend for someone who did not write it. When using continuations for control structures multiple times throughout a program, deciphering their meaning comes down to recognising the patterns in which they are used. The writer on the other hand, will have to write out these patterns repeatedly, unless they are abstracted in some way. Therefore, the second goal of this research is to develop a method to implement user-friendly control structures with continuations; they should be easy to write and easy to comprehend. This method should transform the visual control flow model into language constructs that implement the modelled functionality.

3. SCHEME

The second research question requires that the framework uses a programming language that supports continuations. In addition, the language should have facilities to abstract away the implementation details in order to make the control structures user-friendly. The implementation of the framework will use Scheme [1], which satisfies these requirements. Scheme is a functional language in the Lisp family and features continuations. Scheme's macro systems allow for abstraction of behaviour in cases where regular functions fall short, by allowing the programmer to extend the language with new syntax. The following sections describe continuations, syntactic abstractions and their relevance to this research.

3.1 Continuations

A continuation represents the control state of a computer program. In a running program, the part of the program that is still to be executed is referred to as the *current continuation.* Some programming languages are able to represent continuations as *first-class objects*, which allows them to be created at runtime by capturing the current state. This also means they can be passed to functions and be returned by them, and that they can be assigned to variables. The interface of continuation objects differs per language. In Scheme, continuations must be called with a single argument that is to be processed (or ignored) at the capture point of the continuation.

21st Twente Student Conference on IT, June 23, 2014.

```
(define (print-range from to)
  (call/cc
    (lambda (return)
      (let ((loop (call/cc (lambda (k) k))))
        (display (call/cc
                   (lambda (print)
                     (if (< from to)
                       (print from)
                       (return 'done)))))
        (newline)
        (set! from (+ from 1))
        (loop loop)))))
```

Listing 1. A function to print an ascending range of numbers that utilises continuations

The scope of *undelimited continuations* is the whole program. Such continuations never return after activation, although they may be run multiple times. When an undelimited continuation has finished executing, the whole program has finished. The most common operator to capture continuations like this was introduced by Scheme as `call-with-current-continuation` [1, section 6.4], often abbreviated as `call/cc`. This is in contrast to *delimited continuations*, which have a limited scope and may return a value. In the rest of this paper, a 'continuation' will refer to an undelimited first-class continuation.

3.1.1 Continuation-based Control Structures

Control structures are language constructs that are used to specify the flow of control, i.e. the execution path of the program. Conditions, loops and exceptions are common control structures; the sequencing of subprograms may also be considered as such. The continuation is another control structure, one that is especially suited to implement other control structures.

Listing 1 illustrates the use of continuations with a function that uses them to print numbers in a given interval. First, a continuation is captured and bound to the symbol `return`, this will be used to escape from the function. Because the call to `call/cc` is the only expression in the top level of the function body, calling this continuation will exit the function and return the value it was called with. Second, we capture another continuation to serve as the entry point into a loop. Third, `display` is called to print a number. However, in the case that the whole interval of numbers was printed already, the argument to `display` will not be evaluated to a value and instead the function will return with the symbol `done`. When this is not the case, the number will be printed and incremented.

Note that while `(print from)` may look like a procedure call, it is actually a jump to the continuation bound to the symbol `print` which, intuitively, replaces the call to `call/cc` with the number it was passed. To jump to the start of the loop, the `loop` continuation is applied to itself. This is to ensure that after the jump, `loop` will still refer to the same continuation.

3.2 Abstractions

To repeatedly use a continuation-based control structure without using it as a design pattern, as described by Ferguson and Deugo [2], it is desirable to encapsulate it in some way in order to abstract away the implementation details. In the following sections we will describe three abstraction mechanisms.

3.2.1 Lambda Abstractions

In Scheme, the *lambda abstraction* is the most commonly used abstraction of behaviour; it evaluates to a *procedure*, also called a *function*. However, the scoping of a lambda abstraction, i.e. the local variables and the closure over the static environment, complicates its usage as a control structure. For example, because Scheme is lexically scoped and has a call-by-value evaluation strategy, atomic values (e.g. integers) in the (directly enclosing) dynamic scope of a procedure call cannot be modified by side-effect. To illustrate this shortcoming, say that we want to implement the pre-increment operator from the C programming language (`++i`) in Scheme. This can be achieved with the expression `(begin (set! i (+ i 1)) i)`. Encapsulating this expression into a procedure so we can use something like `(++ i)` will not work, because `set!` would mutate a local copy of `i`.

```
(define-syntax ++
  (syntax-rules ()
    ((++ loc)
     (begin (set! loc (+ loc 1)) loc))))
```

Listing 2. The pre-increment operator implemented as a macro

Another issue is that the programmer cannot control the evaluation of procedure arguments from the procedure body. Instead, they are always evaluated before the procedure body is executed, in an unspecified order. When implementing control structures it is necessary to have control over the evaluation of certain elements, such as branches in a conditional construct, so that evaluation is only done when it is necessary and in the right order. There are ways to overcome this limitation though, for example by using a closure without arguments, a so called *thunk*, to encapsulate an expression that may or may not be evaluated later on. Scheme also has the lazy evaluation constructs `delay` and `force` [1, sections 4.2.5 resp. 6.4] that can be used for this purpose.

3.2.2 Macros

An alternative abstraction mechanism is a *macro transformer*. Scheme macros can be seen as a special kind of function: whereas procedures accept values and return one or more values, macros accept pieces of code[1]and transform them. Macros are more flexible than procedures with regard to scoping while they are also able to reduce the need for boilerplate code. For example, explicit passing of thunks or continuations can be made implicit with macros, increasing readability. This can improve the user-friendliness of the implemented control structures.

The pre-increment operator mentioned earlier can be implemented by using Scheme's `syntax-rules` macro system [1, section 4.3], as shown in Listing 2. Because the `++` expression is replaced by the `begin` expression before the code is evaluated, no new scope or local copy of `loc` is created and the operator works as intended. As an example of a macro simplifying an existing construct, consider Listing 3, which uses `call-with-values`, Scheme's primitive for binding identifiers to values from a multi-valued expression[2][1, section 6.4]. `produce` returns three values that are passed on to `consume`, whose return value will be the result of `call-with-values`. Although `call-with-values` can accept named procedures as arguments, the procedures are usually passed as anonymous lambda expressions. The essence of this mechanism consists of the two procedure bodies and the list of binding names; the `receive` macro used in Listing 4 implements an alternate

[1]More formally: *S-expressions*, the list structure representing Lisp programs.

```
(call-with-values
  (lambda () (produce))
  (lambda (v1 v2 v3) (consume v1 v2 v3)))
```

Listing 3. Handling multiple return values using the Scheme primitive `call-with-values`

```
(receive (v1 v2 v3) (produce)
  (consume v1 v2 v3))
```

Listing 4. Handling multiple return values using the `receive` operator, as described in Scheme Request for Implementation 8 [6]

syntax with only these essential parts. This specific use of `receive` will *expand*, i.e. be syntactically transformed, into the expression in Listing 3.

A disadvantage of Scheme macros, when compared to procedures, is that they are not first-class. Some dialects of Lisp do have first-class macros called *fexprs*, but they arguably have more downsides than plain macros [3]. While this means that macros cannot be passed around, we currently do not know of a scenario where this will be an issue in the context of control structures.

3.2.3 Code as Data and `eval`

Macros and predefined syntactic forms such as `lambda` and `define` have a special evaluation strategy that restricts the ways they can be used. For example, the variant of `define` that assigns a value to a (new) variable, e.g. `(define foo 42)`, does not evaluate its first argument but takes a symbol as-is. This means that normally, the variable name cannot be determined at runtime and has to be hard-coded. We can get around this restriction by using `eval`.

The Scheme procedure `eval` takes a Scheme expression represented as a list structure and an environment and evaluates the expression in the environment [1, section 6.5]. The environment provided by the procedure `interaction-environment` contains the variable and syntax bindings described by the standard and also the top-level bindings of the running program. For example, `(eval (list '+ 1 2))` represents the expression `(+ 1 2)` and evaluates to 3. With `eval` we can thus evaluate code that has been generated at runtime, because Scheme code can be represented by data, i.e. regular Scheme lists. Another advantage is that we can manipulate the symbols used in the code itself. For example, the procedure in Listing 5 defines a variable with a name from a string.

When building expression lists using `list` and `cons`, it becomes quite hard to figure out what the modelled code is supposed to look like. For a more compact syntax, quotation of lists can be used so that the expressions look like normal code. One could then define a variable by evaluating the expression `'(define foo 42)` instead of `(list 'define 'foo 42)`. However, quoting comes with the disadvantage that the list cannot be constructed at runtime, so the used symbols have to be hard-coded. `quasiquote` combines the advantages of `list` and `quote`: a quasiquoted list is constant, except for the parts that are *unquoted* into it. Using `quasiquote`, the `define` expression from Listing 5 would look like this[3]:

```
`(define ,(string->symbol name-string) ,value)
```

[2]Using the `values` procedure, procedures can return multiple values at once without using a data structure to compose them.

```
(lambda (name-string value)
  (eval (list 'define
              (string->symbol name-string)
              value)
        (interaction-environment)))
```

Listing 5. A procedure defining a top-level variable with a name that is only known at runtime

```
(lambda args
  (apply + args))

(lambda args
  (eval `(+ ,@args) (interaction-environment)))
```

Listing 6. Two ways of expressing a variadic function summing its arguments

A related feature is `unquote-splicing`, which unpacks a list into the surrounding quasiquoted list. An empty list spliced into a quasiquoted list does not add any elements. For an example usage of `unquote-splicing`, consider the two procedures in Listing 6 which looks at `unquote-splicing` and `apply` side by side.

3.3 Abstractions in our Implementation

Our custom control structures are implemented by lambda abstractions. `quasiquote`, `unquote` and `eval` are used extensively for the generation of these procedures. By letting the user specify the behaviour of a control structure instance (e.g. the body of a loop) using thunks, our implementation has control over the order of evaluation. Furthermore, because thunks can close over the dynamic environment, the user will be able to modify variables in the enclosing scope. Macros are then used to make the control structures less verbose. The implementation is discussed further in section 6.

4. VISUAL MODEL

In this section we will describe the visual model that can be used to describe control structures. After modelling a control structure in this way, we can translate it unambiguously to a Scheme data structure, which will be described in section 5. Any translation from the Scheme representation to the visual model will also be unambiguous, so that there is a one-to-one correspondence. The representation in Scheme will then be analysed by a program which will provide us with an actual implementation of the modelled control structures; this will be described in section 6.

4.1 Code Bodies and Jumps

We will illustrate the visual model with a couple of control structures. We first examine the subroutine control structure in Figure 1. The name is displayed on top and is part of the model. Scheme procedures return the value of the last expression in the body; there is no standard way to return early, but this can in fact be done by using continuations. The big circle in the figure represents the behaviour, or *code body*, that is to be executed. The arrow pointing into this circle has a smaller circle attached to it which denotes that before execution of the body, the program state is saved into a *continuation slot* with the number 0.

[3]`'`, `` ` ``, `,` and `,@` are syntactic sugar for the syntactic keywords `quote`, `quasiquote`, `unquote` and `unquote-splicing`, respectively.

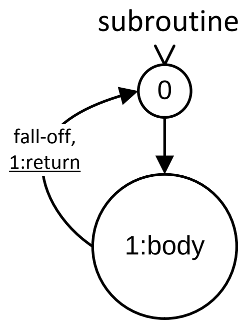

Figure 1. Visual model of a subroutine

The caller of the control structure is represented by the wedge pointing to continuation 0. Continuations occupy a slot numbered with an integer, with 0 always representing the state before entering the control structure. We will refer to continuations like this as the *entry continuation.* The arrows represent *jumps*, the flow of control between code bodies and/or continuations. The arrow from the body to the entry continuation has two labels attached to it, which denote *triggers*. `fall-off` denotes the normal end of execution, i.e. the full evaluation of the body. The `return` label denotes an (early) return from the attached code body, triggered by use of the *keyword* `return` inside of the body.

Keyword labels are underlined to differentiate them from standard triggers such as `fall-off`. This means that, while `fall-off` has a special meaning within the model, it could still be used as a user-defined keyword by underlining it, if one so desires. The integer prefix for `body` and `return` is an implementation detail and will be explained in section 6. When either of these two triggers is activated, execution continues at the target of the jump. In this case, the saved state of continuation 0 is restored and execution continues outside of the control structure. When the jump is triggered explicitly by `return`, it may optionally pass a value back to the caller of the control structure. When the jump is triggered implicitly by fall-off, the value of the last evaluated expression is passed, like in normal Scheme procedures.

Note that while the arrows in the diagram may seem to suggest a loop in the control flow, there is no actual loop. The arrow pointing out of the small circle denotes a capture of a continuation, while the arrow pointing into it means that the saved continuation is called in order to continue control flow in the *context of the continuation.* Continuing in this context means that the underlying call to `call/cc` returns with the value that was passed to the continuation. The context of the capture belongs to the caller of control structure and is not explicitly shown in the diagram.

4.2 Conditional Jumps and Lexical Keywords

For a more complex control structure, let us consider the for loop in Figure 2. This control structure has multiple code bodies with different names, which correspond to the C-like syntax `for (<initialiser>; <terminator>; <incrementor>) body`. After the `initialiser` and the `terminator`, there are two jumps labeled `true` and `false`. While `fall-off` represents an *unconditional jump*, these labels represent *conditional jumps.* The choice of the jump is based on the value of the last evaluated expression in the attached body of code: `false` will be triggered if the last value is `#f`, otherwise the `true` branch will be taken,

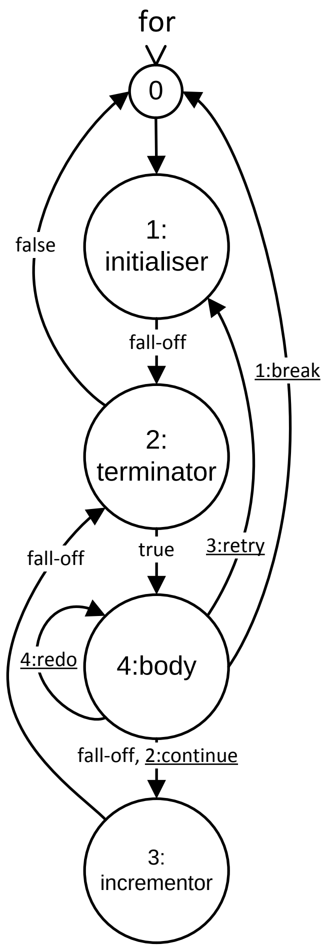

Figure 2. Visual model of a for loop

analogous to standard conditional constructs in Scheme such as `if` and `cond`. In our implementation, the tested value is also implicitly passed into the next body as an argument; the details of this will be covered in section 6. Each code body should have exiting jumps labeled with either `fall-off` or both `true` and `false`.

Besides the usual `break` and `continue` keywords, this model also includes the `retry` and `redo` keywords from the Ruby programming language, which will resume execution at the beginning of the whole loop or the current iteration, respectively. Note that these keywords are local to the code body labelled `body` only and cannot be used outside of the control structure, nor in the other code bodies. They are *lexically scoped*, which means that they can only be used directly in the code body.

4.3 Dynamic Keywords

The try/catch clause modelled by Figure 3 shows that in addition to the entry continuation, a second one is being saved whenever an exception is raised with the `raise` keyword. Note that `raise` has no number in front of it. The precise meaning of these numbers will become clear in section 6. For now, it suffices to say that lexical keywords have a prefix, and the keywords that do not are *dynamically scoped.* This means that while the attached code body is being executed, in this case `try`, `raise` may be used to raise an exception.

When `raise` is used outside of this context, an error will be thrown at runtime. The context will be lost when fall-off occurs or an exception is raised. When a try/catch

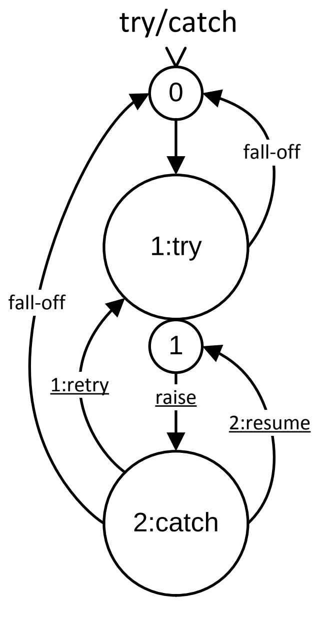

Figure 3. Visual model of a try/catch clause

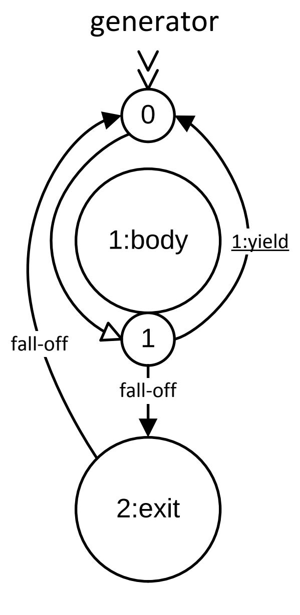

Figure 4. Visual model of a generator

clause is executed within the context of another, the context switches to the inner (most recent) try clause until it is exited. Because of this mechanism, raising an exception within the inner catch clause will cause it to be caught by the outer catch clause. (Re)raising an exception within the most outer catch clause however, will result in an error. One can think of this as a *stack of contexts*: when entering a `try` body a new context is pushed onto the stack, and while propagating an exception the stack is popped until it is either caught or an error is thrown when the stack is depleted.

4.4 Reentrant Control Structures

The generator control structure in Figure 4 differs from the previous examples in that it is *reentrant*. This means that, unlike the control structures that are 'fresh' every time they are executed, this one keeps its own state between invocations. This is indicated by the double wedge pointing to continuation 0. The state is saved in the form of continuation 1, which is indicated by the arrow with the white arrowhead pointing into it. Continuation 1 would normally hold the state saved on fall-off or when using the `yield` keyword. However, on the first invocation `body` will not have run yet, so the continuation will be uninitialised and execution will transfer to the beginning of `body`. In this context, continuation 1 is called a *guarded continuation.*

After yielding a value, the generator can be called again and again, until `body` is done executing. At that point, the state will be saved a final time, and `exit` will execute, which should return a value indicating that there are no more values to generate. Generator calls after that will also cause `exit` to be called. In our implementation, the interface for making reentrant control structures is different from non-reentrant ones because of the inherent state; this will be covered in section 6.

All the features of the visual model have been shown in the previous examples. The next section will give a more formal specification of the model, the way it is represented in the programming language itself.

5. LANGUAGE-EMBEDDED SPECIFICATIONS

To process the control structure specifications in Scheme, we have developed a format which can be used to specify the control structures in the language itself, as a list structure. The translation between the visual model and the *language-embedded specification* is unambiguous and no extra information is added or lost in the process. Figure 5 shows the grammar of this representation in a BNF notation, where ⟨control-structure-spec⟩ is the starting rule. Words prefixed with a quote represent literal symbols. The nonterminals ⟨boolean⟩, ⟨integer⟩ and ⟨symbol⟩ represent values of the indicated Scheme types. Listing 7 shows the specifications of several control structures using this grammar, corresponding to the visual models in Figures 1, 2, 3 and 4.

5.1 Name and Entry

We will now explain the semantics and constraints of the specification data structure. When "is/are" or "shall (not)" are used to describe part of a specification, this is to be interpreted as a requirement. Violating a requirement may or may not cause errors to be thrown or undefined behaviour to occur while constructing the control structure or while executing it.

For the first list element, the control structure needs a symbol representing a name. This name is used for the variable for the procedure implementing the control structure, as described in section 6, and also for any error messages thrown by the procedure. The second element is a list with two pieces of information. Firstly, there is a symbol declaring whether or not the control structure is reentrant or not. Control structures that remain state between invocations such as generators and coroutines should be declared as reentrant, the exact details of this will be discussed in section 6.

Secondly, there is an optional element specifying where the control structure should begin its execution. When no starting point is specified, it is taken to be the code body that is mentioned first, as described in the next paragraph. Reentrant control structures shall start either at a code body or at a guarded continuation. A guarded continua-

```
⟨control-structure-spec⟩ ::= (⟨name⟩ ⟨entry⟩ ⟨code-bodies⟩)
⟨name⟩                   ::= ⟨symbol⟩
⟨entry⟩                  ::= ⟨non-reentrant⟩
                           | ⟨reentrant⟩
⟨non-reentrant⟩          ::= ('non-reentrant ⟨body-name⟩?)
⟨body-name⟩              ::= ⟨symbol⟩
⟨reentrant⟩              ::= ('reentrant
                             ⟨reentry-destination⟩?)
⟨reentry-destination⟩    ::= ⟨body-name⟩
                           | ⟨guarded-continuation⟩
⟨guarded-continuation⟩   ::= (⟨body-name⟩
                             ⟨continuation-id⟩)
⟨continuation-id⟩        ::= ⟨integer⟩
⟨code-bodies⟩            ::= (⟨code-body⟩+)
⟨code-body⟩              ::= (⟨body-name⟩ ⟨jump⟩+)
⟨jump⟩                   ::= (⟨trigger⟩ ⟨destination⟩
                             ⟨saved-continuation⟩?)
⟨trigger⟩                ::= ⟨fall-off-trigger⟩
                           | ⟨keyword-trigger⟩
⟨fall-off-trigger⟩       ::= ⟨unconditional-falloff⟩
                           | ⟨conditional-fall-off⟩
⟨unconditional-fall-off⟩ ::= 'fall-off
⟨conditional-fall-off⟩   ::= ⟨boolean⟩
⟨keyword-trigger⟩        ::= (⟨scope⟩ ⟨keyword⟩)
⟨scope⟩                  ::= 'lexical
                           | 'dynamic
⟨keyword⟩                ::= ⟨symbol⟩
⟨destination⟩            ::= ⟨body-name⟩
                           | ⟨continuation-id⟩
                           | ⟨guarded-continuation⟩
⟨saved-continuation⟩     ::= ⟨continuation-id⟩
```

Figure 5. Grammar of the language-embedded specifications of control structures

```
(define sub-spec
  '(subroutine
    (non-reentrant)
    ((body (fall-off          0)
           ((lexical return) 0)))))

(define for-spec
  '(for
    (non-reentrant)
    ((initialiser
      (fall-off            terminator))
     (terminator
      (#t                  body)
      (#f                  0))
     (incrementor
      (fall-off            terminator))
     (body
      (fall-off            condition)
      ((lexical break)     0)
      ((lexical continue)  incrementor)
      ((lexical retry)     initialiser)
      ((lexical redo)      body)))))

(define try/catch-spec
  '(try-catch
    (non-reentrant)
    ((try   (fall-off          0)
            ((dynamic raise)   catch 1))
     (catch (fall-off          0)
            ((lexical retry)   try)
            ((lexical resume)  1)))))

(define generator-spec
  '(generator
    (reentrant (body 1))
    ((body (fall-off         exit 1)
           ((lexical yield) 0    1))
     (exit (fall-off         0)))))
```

Listing 7. Specifications of a subroutine, for loop, try/catch clause and generator

tion is tied to a body that will be executed when the continuation is uninitialised. This means that the body will be executed on the first use of the control structure, and on later invocations the continuation may be called instead, if its continuation slot was saved to before. Non-reentrant control structures can only start at a body, because they cannot hold state between invocations.

5.2 Code Bodies

The third and last element is a list of at least one code body specification. The code bodies shall be ordered ascendingly, based on their number prefix in the visual model, which shall be a positive integer used at most once in a single control structure. A code body has a symbol name which shall be unique within the control structure and shall be the first element in the specification. This name is used for the names of generated procedures, as described in section 6.

Following the name are one or multiple jump specifications. Their order is not important, other than that the lexical keywords shall be ordered ascendingly (but not necessarily consecutive), based on their number prefix in the visual model, which shall be a positive integer used at most once for a specific jump. A jump is specified by a trigger, a destination and an optional continuation slot to save to.

A trigger is either unconditional fall-off, conditional fall-off or a keyword. Each body shall have exactly one kind of fall-off behaviour, either conditional or unconditional. Furthermore, a conditional jump shall be represented by exactly one `#t` trigger and one `#f` trigger. A keyword shall have a name and either a lexical or dynamic scope. A specific lexical keyword name shall not be declared more than once in a code body. However, a lexical keyword may be used in different code bodies and its semantics may vary among them, i.e. its destination and continuation-saving behaviour may vary. Each body may have one dynamic keyword associated with it, which may have the same name as a dynamic keyword defined elsewhere. One may for example let different exception-handling control structures share the `raise` keyword, this will mean that they share the same stack of contexts. A dynamic keyword may also take the same name as a lexical keyword regardless of where the lexical keyword is defined (although we see no reason to do this); section 6 will discuss why this is.

The destination of a jump is either a body, continuation or a guarded continuation. For an unguarded jump to a continuation, the continuation is assumed to be initialised. If this is not the case at runtime, an error will be thrown. A body name shall refer to a body within the same control structure and a continuation number shall refer to a continuation slot used somewhere in the same control structure. Slot 0 is reserved for the entry continuation. In the visual model, body and continuation jumps are represented by an arrow with a black head, while guarded continuation jumps have white arrowheads. There may be multiple jumps into a body or continuation, and both unguarded and guarded continuation jumps may be targetting a specific continuation slot; the guard is tied to the jump and not to the continuation.

The last element of the jump is optional and represents a continuation slot to save to. With no slot specified, the jump performs no save. A specified slot shall be a positive integral number. Multiple jumps may save into the same continuation, but a saved continuation shall only be attached to a single code body, i.e. code bodies shall not share the same continuation slot.

6. IMPLEMENTATION

We will now describe how the control structures are implemented, i.e. how the specifications are transformed into procedures representing the control structures. The source code of the implementation consists of four files which can be found on our GitHub repository[4]. `control-structures.scm` contains definitions of procedures that generate the modelled control structures. `macros.scm` contains macros that provide a more convenient interface for the control structures on top of these procedures. `control-structure-specs.scm` contains functions for extracting information out of the specifications and defines a number of control structure specifications, including the ones illustrated in this paper. `misc.scm` contains some utility functions that are not directly related to control structures.

6.1 Procedures

Most of the processing of the control structure specifications is done by the procedure `make-control-structure` in `control-structures.scm`, which takes a control structure specification and returns code as data that represents a procedure that implements the control structure.

We will first describe the interface for non-reentrant control structures. Evaluating the code for a control structure yields a procedure that takes a procedure as argument for every code body, in the order specified by the specification. They are followed by an optional argument that is passed to the first body that is executed. A code body procedure is provided by the user and takes as its first argument a value that is passed in via jumps targeting that specific code body. For the rest of its arguments, the procedure accepts procedures that represent lexical keyword jumps, in the same order as in the specification. These jump procedures are provided by the control structure and accept a single optional argument that is passed to the body or continuation it is targeting.

A control structure procedure returns the value that is passed to the entry continuation in slot 0. A code body procedure returns the value that is passed by a fall-off jump; in case of conditional fall-off this value is inspected to make a choice between the `#t` and `#f` branches. A code body procedure is not required to return, but the specification requires that a fall-off jump is always specified. A lexical keyword jump procedure never returns. A summary of these procedures is given in Table 1.

The interface for reentrant control structures is slightly different. Evaluating the code returned by `make-control-structure` does not yield a control structure directly, but a *constructor* for control structures. This is necessary because each instance of the control structure holds its own state, which is initialised by the constructor. This constructor accepts code body procedures as its arguments, and returns the actual control structure with these code bodies. This control structure is then ready for execution and accepts an optional argument, which is passed to the first code body. After returning a value, the control structure can be called again and make use of any state that was saved at the previous execution(s). The interface for the code bodies and lexical keyword jumps is the same as with non-reentrant control structures. A summary of these procedures is given in Table 2.

`control-structures.scm` also defines `initialise-control-structure`, which also takes a control structure specification as an argument. Calling this procedure has two effects. Firstly, it defines a function called `<name>-proc` that implements the control structure, where `<name>` is the name included in the specification. Secondly, it defines procedures for any dynamic keywords that were not defined previously, and the stacks in which their contexts are stored. The dynamic keyword procedures have the same name as the keyword itself and accept an optional argument that is passed to its target.

While the argument to the first body and to keywords is optional, code bodies always receive an argument, whether or not one was explicitly given. When no value was passed, the argument of the code body is actually a special value called `nothing`, which can be tested for with the predicate `nothing?`. This feature is meant to abstract away the implementation of optional keywords, which is done with a variable argument list.

As an example of the generated code, Listing 8 shows the code generated by calling `(make-control-structure generator-spec)`, with `generator-spec` as defined in Listing 7. `let/cc` is a macro using `call/cc`, it captures a continuation and binds it to the symbol in its first argument. `car-or-nothing` converts the optional argument list into the correct form and checks for surplus arguments.

6.2 Usage

We will illustrate the usage of the control structures with the for loop from Figure 2 and the generator from Figure 4. Listing 9 shows an interpreter session in which a generator `gen` is defined which yields all integers smaller than 15 that are relatively prime to 15, i.e. have no divisors in common except for 1. The body of the generator contains a for loop. The dummy expression `#f` is used in the `initialiser`, because no initialisation is needed. The returned values are shown to the right of the arrows. The generator is repeatedly called until it is exhausted and starts returning the `exit` value.

Listing 10 shows a definition of `relative-primes` in an alternative style that avoids assignment by always passing the counter to the next body. This generator needs an argument when it is called for the first time, which will be the initial value of `i`. Values passed in after the first execution of the generator will be ignored, because `yield` is not placed in a context that uses its return value. The user of the generator may also choose to use `yield`'s return value inside of the generator, which allows it to be used as a *coroutine*. For a more convenient coroutine interface, we could also add an `exit` keyword which stops the execution before the end of the body.

Using the `syntax-case` macro system, as described in the R6RS Standard Libraries [4, section 12], we can make the use of the control structures less verbose. In this case, the simpler `syntax-rules` system, described in R5RS [1], does not suffice because we want the macros to define identifiers that can be used by the programmer, i.e. lexical keywords and argument names. Listing 11 shows macros for the for loop and generator, which remove the need for explicit lambdas. Listing 12 shows what the definition of `relative-primes` may look like when these macros are used. Note that with or without the usage of macros, the lexical keywords have to be named by the user and

[4]`github.com/mipiro/control-structures-framework`

Table 1. Overview of non-reentrant control structure procedures

Procedure	Provided by	Parameters	Return value
control structure	`make-control-structure` (as data)	code body procedures, argument to first body (optional)	value passed to entry continuation
code body	user	jump argument, lexical keyword jumps	fall-off jump value
lexical keyword jump	control structure	jump argument (optional)	does not return

Table 2. Overview of reentrant control structure procedures

Procedure	Provided by	Parameters	Return value
control structure constructor	`make-control-structure` (as data)	code body procedures	control structure procedure
control structure	control structure constructor	argument to first body (optional)	value passed to entry continuation
code body	user	jump argument, lexical keyword jumps	fall-off jump value
lexical keyword jump	control structure	jump argument (optional)	does not return

```
(load "control-structures.scm")    ⟹ unspecified
(initialise-control-structure
  generator-spec)                  ⟹ unspecified
(define (relative-primes num)
  (generator-proc
    (lambda (arg yield)
      (define i 1)
      (for (lambda (arg)
             #f)
           (lambda (arg)
             (< i num))
           (lambda (arg)
             (set! i (+ i 1)))
           (lambda (arg break continue
                        retry redo)
             (if (= (gcd i num) 1)
               (yield i)))))
    (lambda (arg)
      'done)))                     ⟹ unspecified
(define gen (relative-primes 15)) ⟹ unspecified
(gen)                              ⟹ 1
(gen)                              ⟹ 2
(gen)                              ⟹ 4
(gen)                              ⟹ 7
(gen)                              ⟹ 8
(gen)                              ⟹ 11
(gen)                              ⟹ 13
(gen)                              ⟹ 14
(gen)                              ⟹ 'done
(gen)                              ⟹ 'done
```

Listing 9. Interpreter session demonstrating the use of a generator

```
(define (relative-primes num)
  (generator-proc
    (lambda (arg yield)
      (for (lambda (i)
             i)
           (lambda (i)
             (and (< i num) i))
           (lambda (i)
             (+ i 1))
           (lambda (i break continue
                      retry redo)
             (if (= (gcd i num) 1)
               (yield i))
             (continue i))))
    (lambda (arg)
      'done)))
```

Listing 10. Value-passing generator of relatively prime numbers

```
(define (relative-primes num)
  (generator 'done
    (define i 1)
    (for #f (<= i num) (set! i (+ i 1))
      (if (= (gcd i num) 1)
        (yield i)))))
```

Listing 12. Shorter definition of `relative-primes` using macros

are not automatically derived from the specification. This could be done automatically for the macros, but then they would be less flexible with regard to the other elements of the syntax, such as the order of the code bodies.

6.3 Limitations

Certain relations between code bodies cannot be expressed without using code, so they cannot be incorporated into the model. For example, the `catch` body from the control structure in Figure 3 should reraise the passed exception if it cannot handle it. However, the missing logic can be incorporated into a macro, as shown in `macros.scm`.

The modelled control structures have a fixed number of code bodies, which means that control structures with a variable number of clauses like C's `switch` statement cannot be modelled. These control structures would thus have to be coded manually.

A continuation save slot can only be associated with one code body. Allowing multiple bodies to share the same slot would make the model more flexible, but it would not integrate well with the rest of the visual model. If a continuation is represented by multiple small circles, it raises the question whether a jump to it should be represented by one arrow to one of the continuation circles or by arrows to all of them; both options are not very intuitive. If a continuation may only be represented by a single circle, the sharing bodies would have to form a crowd around this circle, and the jumps out of the shared continuation should then always have the same trigger(s), which would be less flexible with regards to the features of the control structure as well as the visual layout of the graph. Moreover, it would also introduce the need for more runtime bookkeeping when dynamic keywords are involved, because their

```
(lambda (user-body user-exit)
  (let ((conts (vector #f
                       (lambda (arg)
                         (error 'generator
                                "continuation #1 was called without being initialised"))))
        (defined-conts (make-vector 2 #f)))
    (lambda args
      (let/cc entry-cont
        (if (vector-ref defined-conts 1)
          ((vector-ref conts 1) (car-or-nothing args))
          (begin
            (vector-set! conts 0 entry-cont)
            (vector-set! defined-conts 0 #t)
            (letrec ((internal-body (lambda args
                                      (let ((result
                                              (user-body
                                                (car-or-nothing args)
                                                (lambda args
                                                  (let/cc k
                                                    (vector-set! conts 1 k)
                                                    (vector-set! defined-conts 1 #t)
                                                    ((vector-ref conts 0)
                                                      (car-or-nothing args)))))))
                                        (let/cc k
                                          (vector-set! conts 1 k)
                                          (vector-set! defined-conts 1 #t))
                                        (internal-exit result))))
                     (internal-exit (lambda args
                                      (let ((result (user-exit (car-or-nothing args))))
                                        ((vector-ref conts 0) result)))))
              (internal-body (car-or-nothing args)))))))))
```

Listing 8. Generated code implementing a generator

context stacks are different for each code body, and the body corresponding to the saved continuation would only be known at runtime.

Because jumps to code bodies are implemented as procedure calls, when there are a lot of jumps they should occur in a tail context, as described in R5RS [1, section 3.5]. With fall-off triggers, this is always the case, but care should be taken with keyword usage in order to avoid stack overflow.

Explicit use of `call/cc` or continuation calls within a control structure may cause undefined behaviour because the system is brought into a state that would otherwise not be possible. This may also be caused by using keywords of other control structures inside a code body with dynamic keywords. For example, when a user escapes from the `try` body of the control structure depicted in Figure 3 using his own continuation, the procedure implementing the `raise` keyword will be left on top of its context stack, which can cause problems when exception handling is needed at a later occasion. To avoid this problem, all uses of `call/cc` should be encapsulated by generated control structures. Moreover, in the code bodies with a dynamic keyword one should only use the keywords specified by that control structure and not those of others. This problem might be solved by using Scheme's `dynamic-wind` [1, section 6.4]; this requires further research.

The scope of variables introduced in code bodies cannot be specified, as this would complicate the model. This means that a variable introduced in the `initialiser` of a for loop, as in Figure 2, would not be visible inside of the `body`. The ability to pass a value onto the next body mitigates this somewhat.

While dynamic keywords have global scope, lexical keywords should not be used outside of their code body. However, the procedure that implements the jump can 'leak' out of the body by assigning it to an external variable or by passing it as value to a jump. It would be possible to implement a check that ensures an error will be thrown if an illegal access is made to a keyword procedure, but this would incur an overhead at every jump.

The procedures implementing dynamic keywords should have global scope, but the stacks maintaining their contexts and the list of registered keywords should not be global, although this is the case in the current implementation. This could be solved by using a module system, such as the one described in R6RS [5, section 7], but this is outside the scope of this research.

Loops cannot be assigned labels to jump to with `break` and `continue`, as is possible in Java. Allowing this would complicate the model. This can be solved by using the subroutine control structure together with looping constructs and calling `return` to jump to the desired code section. Alternatively, one may throw an exception to be caught at the desired section of code.

Jumps can be passed at most one argument, this is because the Scheme standards prescribe that continuations are limited to one argument and we decided to keep the interface for body jumps the same as continuation jumps, for simplicity. There are Scheme implementations in which continuations may accept zero or multiple arguments, but implementing the framework for these Scheme versions would sacrifice portability in favour of flexibility.

7. RELATED WORK

Ferguson and Deugo have shown how to use Scheme's continuations as patterns [2]. Our work uses the same underlying principles, but avoids patterns by having the code automatically generated. The downside of our model is that it is less flexible, and the current implementation has a higher runtime overhead because it has little optimisation.

```
(define-syntax for
  (lambda (stx)
    (syntax-case stx ()
      ((for initialiser terminator incrementor body ...)
       (let ((arg      (datum->syntax #'for 'arg))
             (break    (datum->syntax #'for 'break))
             (continue (datum->syntax #'for 'continue))
             (retry    (datum->syntax #'for 'retry))
             (redo     (datum->syntax #'for 'redo)))
         #`(for-proc (lambda (#,arg) initialiser)
                     (lambda (#,arg) terminator)
                     (lambda (#,arg) incrementor)
                     (lambda (#,arg #,break #,continue #,retry #,redo) body ...)))))))

(define-syntax generator
  (lambda (stx)
    (syntax-case stx ()
      ((generator exit-expr body ...)
       (let ((yield (datum->syntax #'generator 'yield))
             (arg   (datum->syntax #'generator 'arg)))
         #`(generator-proc (lambda (#,arg #,yield) body ...)
                           (lambda (#,arg) exit-expr)))))))
```

Listing 11. `syntax-case` macros for the for loop and generator

Te Brinke et al. have stated [7, section III]:

> "We identified at least two requirements for implementing the semantics of control-flow mechanisms as first-class abstractions:
>
> - being able to influence the execution order of statements;
> - having a way to bind different scopes to blocks so as to identify which variables can be used in which context."

With our model, the implementer of the control structure is able to specify both the execution order that is normally taken and also the ways in which the user of the control structure is able to influence this order. We have considered making the scopes of code bodies tailorable, but decided against this because it would complicate the model. Furthermore, we suspect that this would make it unfeasible to implement as first-class objects, i.e. procedures. However, the ability in our model to pass a value on to the next body does cover some use cases for which a tailorable scope would otherwise be desired.

8. CONCLUSION

We have shown how control structures can be modelled visually as a flow graph and defined an equivalent Scheme representation for this specification. The modelled control structures contain a fixed number of executable code bodies that may transfer control by jumping into one another; this can be done automatically, conditionally, or it can be done through explicit request by the programmer, using keywords that either have a lexical or a dynamic scope. Control may transfer to the beginning of a code body or it may start where execution left off previously, through the use of continuations. The model can represent reentrant control structures such as generators. The control structures can be automatically generated from their specifications and their syntax can be defined by the user through the use of macros.

9. REFERENCES

[1] N. Adams IV, D. Bartley, G. Brooks, R. Dybvig, D. Friedman, R. Halstead, C. Hanson, C. Haynes, E. Kohlbecker, D. Oxley, et al. Revised [5] report on the algorithmic language Scheme. *ACM Sigplan Notices*, 33(9):26–76, 1998.

[2] D. Ferguson and D. Deugo. Call with current continuation patterns. In *8th Conference on Pattern Languages of Programs*, 2001.

[3] K. M. Pitman. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on Lisp and functional programming*, pages 179–187. ACM, 1980.

[4] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised [6] report on the algorithmic language Scheme — standard libraries, 2007.

[5] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised [6] report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

[6] J. D. Stone. SRFI 8: receive: binding to multiple values. Online: `http://srfi.schemers.org/srfi-8/srfi-8.html`, 1999.

[7] S. Te Brinke, M. Laarakkers, C. Bockisch, and L. Bergmans. An implementation mechanism for tailorable exceptional flow. In *Exception Handling (WEH), 2012 5th International Workshop on*, pages