Constructing Customized Interpreters from Reusable Evaluators using Game

Stijn Timbermont, Coen De Roover, and Theo D'Hondt

Vrije Universiteit Brussel
{stimberm, cderoove, tjdhondt}@vub.ac.be

Abstract. Separation of concerns is difficult to achieve in the implementation of a programming language interpreter. We argue that evaluator concerns (i.e., those implementing the operational semantics of the language) are, in particular, difficult to separate from the runtime concerns (e.g., memory and stack management) that support them. This precludes the former from being reused and limits variability in the latter.
In this paper, we present the Game environment for composing customized interpreters from a reusable evaluator and different variants of its supporting runtime. To this end, Game offers a language for specifying the evaluator according to the generic programming methodology. Through a transformation into defunctionalized monadic style, the Game toolchain generates a *generic abstract machine* in which the sequencing of low-level interpretational steps is parameterized. Given a suitable instantiation of these parameters for a particular runtime, the toolchain is able to inject the runtime into the generic abstract machine such that a complete interpreter is generated.
To validate our approach, we port the prototypical Scheme evaluator to Game and compose the resulting generic abstract machine with several runtimes that vary in their automatic memory management as well as their stack discipline.

1 Introduction

In the implementation of an *interpreter* for a programming language, one can distinguish evaluator concerns from the runtime concerns that support them. The *evaluator concerns* implement the operational semantics of the language, often assuming computational resources are unbounded. The *supporting runtime concerns* maintain this illusion on a physical machine through data structures and algorithms for memory and stack management.

There is great variation among an interpreter's runtime support, of which we identify the following sources:

– The first source of variation is the evaluator itself. Different language features require different kinds of runtime support. For instance, closures require the ability to capture an environment and keep it alive for an indeterminate period of time. Likewise, exception handling requires the ability to skip computations after an error occurred and to proceed with the exception handler.

These features affect the implementation of environments and of the execution stack respectively.
- The second source of variation is the host platform. If the host platform is a high-level language, it will also have a supporting runtime that can be reused. For instance, when implementing a garbage collected language in another garbage collected language. The bigger the mismatch between the host platform and the evaluator, however, the more effort is required from the supporting runtime. This is, for instance, the case when implementing a language with first-class continuations in C.
- The non-functional requirements of the interpreter represent the third source of variation. The supporting runtime must not only satisfy the needs of the evaluator, it must also do so in a manner that meets efficiency requirements (e.g., memory usage and power consumption). This leads to many trade-offs, which are influenced by the idiosyncrasies of the host platform as well as the expected usage of the interpreter (i.e., the kinds of programs it will run).

On par with the operational semantics of the language, the evaluator should not be affected by variation in runtime support. However, the practice of interpreter development does not reflect this. In the implementation of an interpreter, evaluator concerns are difficult to separate from the runtime concerns that support them. This precludes the former from being reused and limits variability in the latter. The contributions of this paper are as follows:

- Using the case of automatic memory management, we illustrate that the choice for a particular runtime has a severe impact on the structure of the evaluator (Section 2).
- We introduce the notion of a *generic abstract machine* (Section 3.2), an abstract machine [5] that anticipates the supporting runtime without committing to any details using generic programming techniques. A generic abstract machine corresponds to the evaluator in an intermediate form called *defunctionalized monadic style*.
- We present the Game environment (Section 3), consisting of a language and a toolchain centered around the notion of a *generic abstract machine*. Using the Game programming language, the developer of the evaluator decides on the interface between the evaluator and the runtime —thus enabling reuse of the evaluator. Using the Game toolchain, the developer of the supporting runtime can inject a concrete runtime variant in the evaluator —giving rise to a customized interpreter. Our proof-of-concept implementation of the Game toolchain generates this interpreter in a subset of R6RS Scheme. This subset is sufficiently low-level for targeting C to be realistic.
- We validate our approach by implementing the prototypical Scheme evaluator from SICP [1] in Game (Section 3.1) and instantiating the resulting *generic abstract machine* with several runtimes (Section 4). These runtimes vary in their memory management (e.g., a non-moving mark-and-sweep versus a moving stop-and-copy GC) and in their stack discipline (e.g., reusing the host stack versus managing an explicit stack). The generated

interpreters are, together with the GAME prototype, publicly available at `http://soft.vub.ac.be/~stimberm/game/sc12/`.

2 Motivating Example: the Impact of Automatic Memory Management on the Structure of an Evaluator

This section illustrates the impact of runtime support on the structure of the evaluator. Their interaction inhibits reusing the evaluator in other interpreters.

Consider adding automatic memory management to the prototypical evaluator for Scheme depicted in Figure 1. The evaluator corresponds to the one from Section 4.1 of SICP [1], implemented in GAME (cf. Section 3.1). For the purpose of this section, GAME has the same syntax and semantics as regular Scheme.

Garbage collectors (GCs) use reachability as a heuristic to determine whether an object on the heap can be reclaimed. It is the evaluator's responsability to hand the GC all heap objects that are a priori reachable. Objects that cannot be reached from these *root pointers* can no longer be accessed from the program and are therefore safe to reclaim. Root pointer treatment comprises the main source of interaction between an evaluator and a GC.

Varying an interpreter's GC strategy may require varying the treatment of root pointers in the interpreter's evaluator. Some GC algorithms move objects around to avoid fragmentation. All pointers to a particular object must be updated to reflect the new location of the object. Concretely, the evaluator should not use the pointers stored in local variables after every point in its execution where GC may have occurred.

The consequences of errors in the treatment of root pointers are severe. If the evaluator neglects to communicate a root pointer to the GC, the object referred to by the root pointer may be reclaimed. If the evaluator subsequently dereferences this pointer again, the resulting behavior is unpredictable. The memory chunk may have been cleared, or it may have been reused to store another object. After a *moving* GC, a neglected root pointer may even point *in the middle* of some other object. Such bugs are difficult to diagnose. Their occurrence depends on the state of the entire heap and on the arbitrary moment GC occurs.

Restructuring the SICP Evaluator for Garbage Collection As the GC expects to be handed a set of root pointers, we have to restructure the entire SICP evaluator such that it can construct this set at all places where a GC might occur. Consider the `list-of-values` function depicted on line 35 of Figure 1. It returns a list that contains the values to which the expressions `exps` evaluate one by one in the environment `env`. As evaluating an individual expression may trigger a GC, there is a risk of dangling pointers within this function. We will adapt its code such that root pointers are preserved for a non-moving and a moving GC respectively.

Adaptation 1: non-moving GC In Figure 2a, recursive evaluations within the function have been made explicit. In this version, it is clear that variable `first` may contain a root pointer to an object on the heap, which must be preserved

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  ok-symbol)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  ok-symbol)

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (if (last-exp? exps)
      (eval (first-exp exps) env)
      (begin (eval (first-exp exps) env)
             (eval-sequence (rest-exps exps) env))))
```

Fig. 1: The prototypical Scheme evaluator from Section 4.1 of SICP [1]

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (let ((first (eval (first-operand exps) env))
            (rest  (list-of-values (rest-operands exps) env)))
        (cons first rest))))
```

(a) with explicit names for subcomputations

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (begin (register-nm exps)
             (register-nm env)
             (define first (eval (first-operand exps) env))
             (register-nm first)
             (define rest (list-of-values (rest-operands exps) env))
             (unregister-nm 3)
             (cons first rest))))
```

(b) with root registration for a non-moving GC

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (begin (register-m exps)
             (register-m env)
             (define first (eval (first-operand exps) env))
             (define exps2 (unregister-m))
             (define env2  (unregister-m))
             (register-m first)
             (define rest (list-of-values (rest-operands exps2) env2))
             (define first2 (unregister-m))
             (cons first2 rest))))
```

(c) with root registration for moving GC

Fig. 2: Adaptations of `list-of-values`

across subsequent evaluations triggered by the recursive call to `list-of-values`. Variables `exps` and `env` may also contain root pointers as expressions and environments can be stored on the heap. Figure 2b depicts function `list-of-values` as adapted to a particular non-moving GC. Root pointers are registered with the GC through `register-nm` and are subsequently unregistered through `unregister-nm`. These functions behave in a LIFO manner. The parameter to `unregister-nm` indicates that 3 pointers must be discarded from the root set. Note that `rest` does not have to be registered as it is passed to `cons` without any interleaved evaluation.

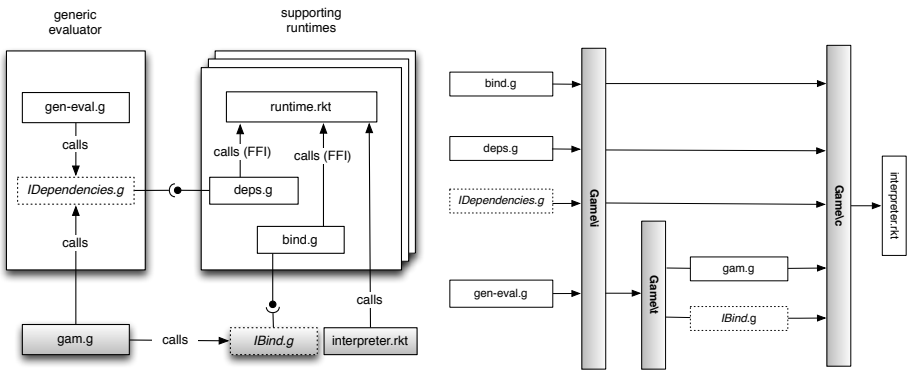
Adaptation 2: moving GC A moving GC further complicates the treatment of root pointers. If a local variable contains a root pointer prior to a potential GC, merely registering that root pointer no longer suffices. The GC may move the object around on the heap and render the pointer stored in the local variable invalid. This problem can be solved by having the GC provide the new root locations to the evaluator, which must then refrain from using the old pointers. Figure 2c depicts function `list-of-values` as adapted to such a moving GC. Root pointers are registered with the GC through `register-m`. In turn, the GC provides the updated root pointer through a corresponding call to `unregister-m`. Again,

these functions behave in a LIFO manner. Note that a root pointer stored in a local variable is never used again after a recursive evaluation.

The two adaptations show that composing the SICP evaluator with a custom GC has a structural impact on the evaluator. Furthermore, the details of the restructuring depends on the chosen GC strategy, which means that it is also not possible to prepare the evaluator for GC once and for all. Instead, the evaluator must be adapted for each variation of memory management. This lack of separation of concerns hinders reuse of the evaluator and evolution of the interpreter as a whole.

3 Overview of Game

The novel notion of a *generic abstract machine* is key to our approach to constructing customized interpreters from a reusable evaluator and different variants of a supporting runtime. A generic abstract machine corresponds to a recursive evaluator implemented using generic programming techniques such that a runtime is anticipated but not yet committed to, transformed into a low-level form that lends itself better to injecting a concrete runtime. To support this approach, we developed Game; the Generic Abstract Machine Environment. Using Game, constructing an interpreter entails four different activities: one for the evaluator developer, another for the runtime developer and two that are left to the toolchain. Before discussing these activities in detail, we briefly outline their key aspects using Figure 3. Figure 3a clarifies the interdependencies of the different kinds of artifacts that are involved in these activities and indicates whether they are generated or have to be provided by one of the developers. Figure 3b depicts how these artifacts flow throughout the environment.



(a) Artifacts (b) Toolchain

Fig. 3: Overview of Game

Developing a generic evaluator (evaluator developer) Game offers the generic programming language Game\l for implementing an evaluator in a high-

level, recursive functional style. Most importantly, this evaluator does not have to adhere to the specifics that would be dictated by the choice for a particular supporting runtime (e.g., the root pointer treatment protocols discussed in Section 2). The evaluator's own dependencies on the runtime (i.e., operations that must be provided by every runtime it will be composed with) are specified through interface declarations. In Figure 3a, `gen-eval.g` and `IDependencies.g` correspond to the generic evaluator and the interface declaration of its dependencies respectively. Note that both artifacts are the sole responsibility of the evaluator developer.

Deriving a generic abstract machine (GAME) Given such a generic evaluator, GAME's transformation engine GAME\t transforms it into an abstract machine —inspired by the work on defunctionalized interpreters [4,2]. The actual transformation inserts hooks that make every computational step in the generic evaluator explicit and programmable. As these hooks are declared through an interface, we call the result a *generic abstract machine*. The transformation requires type information computed for GAME\l input by a type inferencer called GAME\i. In Figure 3a, `gam.g` and `IBind.g` correspond to the generic abstract machine and the interface for the hooks respectively. Note that the generic abstract machine inherits the dependencies of the original evaluator, hence the arrow from `gam.g` to `IDependencies.g`. Figure 3b indicates that `gam.g` is produced from `gen-eval.g` by GAME\t.

Instantiating the generic abstract machine (runtime developer) It is now up to the runtime developer to inject a concrete runtime into the generic abstract machine that was generated above. This will yield a customized interpreter expressed in GAME\l. Note that the runtime developer does not have to analyze the generic abstract machine itself, but only has to implement the interface dependencies of the machine. In other words, `gam.g` can be considered a black box. In Figure 3a, `deps.g` implements the interface of the evaluator dependencies declared in `IDependencies.g`; `bind.g` implements the interfaces for the additional dependencies of the generic abstract machine declared in `IBind.g`. Both implementations import runtime functionality from `runtime.rkt` using a simple Foreign-Function Interface. Developing these three files is the sole responsibility of the runtime developer.

Generating an executable interpreter (GAME) The final activity entails compiling the GAME\l interpreter constructed above to the host platform. In this case, GAME's compiler GAME\c targets R6RS Scheme, in particular Racket[1]. GAME\c performs simple optimizations, converts polymorphic into monomorphic code and removes the overhead introduced by the generic programming. Furthermore, the generated code does not use Scheme's more advanced features such as dynamic typing, long-lived closures, first-class continuations or tail-call elimination in such a way that C code generation would be unfeasible. Figure 3b illustrates how GAME\c compiles the generated `interpreter.rkt` together with `runtime.rkt` into a final executable interpreter.

[1] `http://racket-lang.org/`

```
(define (true? x)
  (not (eq? x false_sv)))
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
(define (lambda? exp)
  (tagged-list? exp lambda-symbol))
(define (lambda-parameters exp)
  (cadr exp))
(define (lambda-body exp)
  (cddr exp))
```

(a) Fragment of the helper functions implemented in Game\l

```
(type-function (SV) *)

(interface (Scheme)
 (ok-symbol (tf SV))
 ...
 (true_sv  (tf SV))
 (false_sv (tf SV))
 ...
 (pair?   (-> ((tf SV)) (effect) Bool))
 (number? (-> ((tf SV)) (effect) Bool))
  ...
 (car (-> ((tf SV)) (effect IO) (tf SV)))
 ...
 (cons (-> ((tf SV) (tf SV))
           (effect IO GC)
           (tf SV)))
 ...)
```

(b) Interface for the required dependencies of the SICP evaluator

Fig. 4: Supporting functions for the SICP evaluator

3.1 Developing a Generic Evaluator

Figure 1 depicts the prototypical Scheme evaluator from SICP [1] implemented in Game\l. Other than substituting `ok-symbol` for the Scheme symbol `'ok`, the code is identical to the original. The evaluator relies on various helper functions, such as `variable?` and `lambda?` for testing whether an expression is of a certain type, and `lookup-variable-value` and `extend-environment` for manipulating environments. Some of these helper functions can also be implemented in Game\l itself (see Figure 4a). However, at some point the evaluator requires functions such as `number?` and `car` which are not available in Game\l. Enumerating these functions as required dependencies renders the evaluator *generic*.

Figure 4b depicts an extract from the Game\l declaration of the required dependencies for the generic SICP evaluator, using an `interface` (akin to a Haskell type class). Note that the required dependencies are explicitly typed. We introduce a single type `sv` for typing Scheme values as Scheme is dynamically typed. Because the concrete type depends on the runtime the evaluator is instantiated with, we define `sv` as a type function [16], without any arguments. The `*` indicates the kind of `sv`. Types can use this type function using `(tf SV)`. For example, the type for `ok-symbol` is simply `(tf SV)`, because symbols are regular Scheme values. The type for the function `pair?` is `(-> ((tf SV)) (effect) Bool)`, where `->` is a type constructor with three arguments; first a list of types for the parameters of the function; third the result type of the function; second the side-effects of the function. Section 3.2 explains the crucial role these effect annotations play.

3.2 Deriving a Generic Abstract Machine

As illustrated in Section 2, the original SICP evaluator has to be restructured before it can be composed with a custom GC. Furthermore, there are variations in the details of the restructuring for a moving and a non-moving GC. Game introduces defunctionalized monadic style to capture the essence of this restructuring and to abstract over the details. As its name suggests, defunctionalized monadic

style derives from monadic style, which generalizes continuation-passing style. Computations are explicitly sequenced, using higher-order functions to represent continuations. Defunctionalized monadic style turns these continuations into explicit data structures, such that the transfer of data between sequenced computations also becomes explicit. Therefore, an evaluator written in defunctionalized monadic style has a more low-level structure than a recursive evaluator.

Figure 5 depicts function `list-of-values` (cf. Figure 1) in defunctionalized monadic style. The essential construct in defunctionalized monadic style is special operator `>>>` (pronounced *bind*). It explicitly sequences two computations, of which the first is passed as the first argument. The second and third parameters represent the second computation. The second parameter is a reference to a continuation function, a top-level function that receives the result of the evaluation. The third argument lists all additional data that should be preserved because it is still needed in the continuation function. The continuation function receives two arguments: first the preserved data, then the result of the evaluation. In the example of `list-of-values`, the first expression in `exps` is evaluated, but both `exps` and `env` are preserved for later, when they are needed in the continuation function `cnt-list-of-values-1`. This is achieved by passing the tuple `(* exps env)` as third argument to `>>>`. That continuation function `cnt-list-of-values-1` uses pattern matching, a GAME\l feature, on its first argument to retrieve `exps` and `env` again. The second argument `first` is the result of evaluating the first operand. The body of `cnt-list-of-values-1` uses `>>>` again to sequence the evaluation of the rest of the arguments with `cnt-list-of-values-2`. This time, only `first` must be preserved. Finally, `cnt-list-of-values-2` simply combines the results using `cons`.

An evaluator expressed in defunctionalized monadic style has the structure of an abstract machine [5], and is better suited for composition with a supporting runtime. Using GAME, abstract machines do not have to be developed by hand. Instead, GAME derives them from evaluators by transforming the latter to defunctionalized monadic style. This transformation is referred to as GAME\t and follows the work of defunctionalized interpreters [4,2], which in turn goes back to the seminal work on definitional interpreters [13]. The essence of their derivation is CPS transformation followed by defunctionalization, yielding first-order, tail-recursive evaluators, which are equivalent to well-known abstract machines. GAME\t works similarly, but adapts the original in two ways: first, GAME\t

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      null
      (>>> (eval (first-operand exps) env)
           cnt-list-of-values-1
           (* exps env))))
(define (cnt-list-of-values-1 (* exps env) first)
  (>>> (list-of-values (rest-operands exps) env)
       cnt-list-of-values-2
       (* first)))
(define (cnt-list-of-values-2 (* first) rest)
  (cons first rest))
```

Fig. 5: The function `list-of-values` in defunctionalized monadic style

```
(defmacro (>>> e k frame)
  (bind (lambda () e) k frame))

(interface (Frame (frm *) (t *) (res *))
  (bind (forall ((e !*) (ke !*))
          (-> (  (-> () e t)
                 (-> (frm t) ke res)
                 frm)
              (effect IO e ke)
              res))))
```

Fig. 6: Declaration of `>>>` and `bind`

generates a *generic* abstract machine, and second, Game\t works *automatically*, whereas in [13,2], the derivation is performed by hand. These two adaptations are further discussed in the following two paragraphs, respectively.

Generic abstract machines A generic abstract machine is expressed in defunctionalized monadic style, where the semantics of `>>>` is customizable, that is, `>>>` is a hook in the abstract machine, used to inject the supporting runtime (see Section 3.3). This is achieved by using generic programming, in the same way that the evaluator dependencies are treated. However, `>>>` is not a regular function because it should not execute its first argument immediately; instead, we define `>>>` as a macro in terms of a function `bind`, by wrapping the first argument to `>>>` in a thunk, as shown in Figure 6. The function `bind` is overloaded on the types of the frame, the value produced by the thunk and the final result of the continuation. The type of `bind` is polymorphic in the effects of the higher-order arguments, hence the effect variables `e` and `ke` with kind `!*`.

Automatic derivation It is not necessary to transform every function call in the evaluator to defunctionalized monadic style. Helper functions such as `number?` and `car` do not trigger a GC, so introducing a `>>>` construct would be overkill. Therefore, Game\t applies the transformation selectively: only computations that potentially trigger a GC are transformed into defunctionalized monadic style. To distinguish between computations that may trigger a GC and those who do not, Game\t uses information derived from a type and effect system in the style of [18]. Effect annotations are useful to track properties such as "may trigger a GC" because they propagate through the evaluator: a function that calls another function which potentially triggers a GC inherits this property. Using Game, there is no need to manually annotate an evaluator with effects. Instead, Game\i performs type and effect inference which derives the required information from unannotated code. However, the declaration of the dependencies must explicitly declared their type as they cannot be "guessed" by Game\i. Figure 4b shows some of the type annotations. Function types not only specify the types of the arguments and result, but also the effect of the function. An effect `(effect ...)` denotes the union of elementary effects. For example, `(effect)` denotes the empty effect and `(effect IO)` the singleton effect, where `IO` indicates general side effects such as accessing and modifying heap-allocated memory. The type of `cons` is special: the effect of the function is `(effect IO GC)`, which means that

```
(type-decl RV () *)
(foreign-import "pair?" r_pair?
  (-> (RV) (effect) RV))
(foreign-import "number?" r_number?
  (-> (RV) (effect) RV))
...
(foreign-import "car" r_car
  (-> (SV) (effect IO) SV))
(foreign-import "cdr" r_cdr
  (-> (SV) (effect IO) SV))
...
(define (bindR thunk k frame)
  (k frame (thunk)))
```

(a) FFI imports and definitions

```
(axiom () () (~ (tf SV) RV))
(axiom () ()
       (? Scheme)
  (mkScheme ...
            r_pair?
            r_number?
            ...
            r_car
            r_cdr
            ...))
(axiom ((frm *)) ()
       (? (Frame frm RV RV))
  (mkFrame bindR))
```

(b) Instantiation with `axiom`

Fig. 7: Default instantiation of the SICP generic abstract machine

`cons` may produce both the general effect `IO` and the effect `GC`. This annotation is propagated through the evaluator and is used by the transformation to distinguish between functions that may trigger a GC and those who do not.

These two adaptations, with the combination of GAME\i with GAME\t, forms the basis for the automatic derivation of generic abstract machines from evaluators. The type annotation for `cons` in Figure 4b indicates that invocations of `cons` are a source of GC. This information is propagated through the unannotated evaluator, such that GAME\i assigns the following type to `eval`.

```
(forall () (Scheme)
        (-> ((tf SV) (tf SV)) (effect IO GC) (tf SV)))
```

The effect annotation `(effect IO GC)` indicates that `eval` may trigger a GC. The type constraint `Scheme` accounts for the required dependencies used in the evaluator. Subsequently, GAME\l introduces `>>>` only for those computations that include `GC` in their effect. This is reflected in the following new type of eval.

```
(forall () (Scheme
             (Frame (*                ) (tf SV) (tf SV))
             (Frame (* (tf SV)        ) (tf SV) (tf SV))
             (Frame (* (tf SV) (tf SV)) (tf SV) (tf SV)))
        (-> ((tf SV) (tf SV)) (effect IO GC) (tf SV)))
```

The three `Frame` constraints arise because `>>>` is overloaded on the type of the frame (via `bind`). For example, the constraint `(Frame (* (tf SV) (tf SV)) (tf SV) (tf SV))` indicates that at some point a `>>>` construct is used to preserve two Scheme values.

3.3 Instantiation of Generic Abstract Machines

In this section we illustrate how we can instantiate a generic abstract machine by providing implementations for both the required dependencies of the original evaluator and the additional dependencies introduced by the derivation of the generic abstract machine. In this section we only give a "default" instantiation to explain the mechanism in GAME. The more interesting instantiations for GC are given in Section 4.

Figure 7 gives the GAME\l code that instantiates the generic machine produced by GAME\t. As the type of `eval` indicates, this instantiation must satisfy four constraints, one `Scheme` constraint and three `Frame` constraints. In Figure 7a,

we use a Foreign-Function Interface (FFI) to import the appropriate functions from the underlying platform, in this case the R6RS Scheme implementation of PLT Racket. We also introduce a single type `RV` denoting a Racket value. In Figure 7b, we use `axiom` constructs (akin to Haskell's type class instances) to actually instantiate the generic abstract machine by satisfying its dependencies. The type function `SV` is instantiated with `RV` and the `Scheme` constraint with the imported functions. The three `Frame` constraints are all instantiated with `bindR`, which simply executes `thunk` and passes the resulting value along with the frame to the continuation. Note that this implementation introduces no supporting runtime, and simply reintroduces the recursion that was made explicit by GAME\t. The `axiom` declaration for `Frame` is polymorphic in the type of the frame, which is why there is only one.

4 Evaluation

To evaluate GAME we instantiate the generic abstract machine derived from the SICP evaluator with several runtimes, which vary in their memory management and stack discipline. Concretely, we give three instantiations: one for a non-moving mark-and-sweep and another for a moving stop-and-copy GC which both rely on the recursion stack of the underlying platform (Racket); and finally an instantiation for a custom stack, using trampolining. We stress again that all three instantiations reuse the original evaluator from Figure 1.

Instantiation for a non-moving mark-and-sweep GC For this instantiation, the implementation for `cons`, imported via the FFI in Figure 7a, is not the standard Racket `cons`, but a custom implementation which uses a mark-and-sweep GC. The code for the instantiation itself is shown in Figure 8b. Figure 8a defines helper functions for the registration of root pointers in frames, which ultimately rely on `register-nm` and `unregister-nm` which were also used in Section 2. These functions are part of the mark-and-sweep memory manager and are also imported using the FFI. In Figure 8b, `bindNonMovGC` satisfies the `Frame` constraints, by registering the roots in the frame before executing `thunk`, then unregistering the corresponding number of roots and finally proceeding with the continuation function. The `axiom` for `Frame` is again polymorphic, by referring to the `NMRoots` interface, because `bindNonMovGC` is defined for every frame that supports `register-roots-nm`.

Instantiation for a moving stop-and-copy GC For a moving GC, the instantiation with `bindNonMovGC` in Figure 8b is not suitable, because the continuation function receives the old `frame`, whose root pointers may be invalidated if a moving GC occurs in the course of executing `thunk`. Figure 9 shows an adapted instantiation. The roots are treated differently, using the function `register-m` and `unregister-m`, which work in a LIFO manner (see Section 2). The function `bindMovGC` in Figure 9b registers to roots before executing the `thunk` and then unregistering them again, which gives the update frame `frame2`, which is then passed to `k`. Note that the original `frame` is not used after registering it.

```
(interface (NMRoots (t *))
  (register-roots-nm
      (-> (t) (effect IO) Int)))

(foreign-import "register-nm" register-nm
  (-> (RV) (effect IO) Unit))
(foreign-import "unregister-nm" unregister-nm
  (-> (Int) (effect IO) Unit))

(define (register0 (*)) 0)
(define (register1 (* val))
  (register-nm val)
  1)
(define (register2 (* val1 val2))
  (register-nm val1)
  (register-nm val2)
  2)
```

(a) Non-moving root registration

```
(define (bindNonMovGC thunk frame k)
  (let* ((cnt (register-roots-nm frame))
         (val (thunk)))
    (unregister-nm cnt))
    (k frame val))

(axiom ((frm *)) ((? (NMRoots frm)))
       (? (Frame frm RV RV))
  (mkNMRoots bindNonMovGC))
(axiom () () (? (NMRoots (*)))
  (mkNMRoots register0))
(axiom () () (? (NMRoots (* RV)))
  (mkNMRoots register1))
(axiom () () (? (NMRoots (* RV RV)))
   (mkNMRoots register2))
```

(b) Instantiation

Fig. 8: Instantiation for a non-moving mark-and-sweep GC

Instantiation for an explicit stack Figure 10 shows an instantiation which maintains an explicit recursion stack instead of reusing the host stack. The definition of `bindStack` in Figure 10b pushes both the continuation and the frame on the stack and simply returns the value of executing the thunk. The function `engine` defines a loop which pops the top continuation of the stack and executes it until the stack is empty. The continuation pushed on the stack in `bindStack` is not the original `k`. Instead, it is wrapped in a `lambda` such that it pops its own frame from the stack, as the layout of the frame may vary across continuation functions. Note however that `k` is a reference to a top-level function and that the `lambda` expression does not capture other local variables. This means that if Game\c inlines `bindStack`, the `lambda` can be lifted, such that again only top-level function pointers are required from the host platform. The functions `push0`, `pop0`, ..., are similar to the `register` functions of the previous instantiation, but instead they ultimately rely on `push-rv` and `pop-rv`. This instantiation effectively converts the evaluator into trampolined style [7]. Therefore, this instantiation would be a good starting point for extending the interpreter with first-class continuations.

Discussion All three instantiations[2] successfully compile to Racket and correctly execute a small Scheme program. In particular, the GC instantiations do not suffer from dangling or corrupt pointers after a GC. Manual instantiations would require three rewrites of the SICP evaluator, whereas by using Game, the evaluator can be reused as is. However, Game also has a number of limitations. First of all, an instantiation of a generic abstract machine consists of a single monolithic entity, which must be composed manually by the runtime developer. For example, code that does not go through the Game toolchain still has to track roots itself. Second, the evaluator is treated as a single entity, with little support for modularization, especially compared to modular interpreters using monad transformers [11,17]. However, we believe both of these concerns to be

[2] The full code of the generated interpreters can also be found on `http://soft.vub.ac.be/~stimberm/game/sc12/`.

```
(interface (MRoots (t *))
  (register-roots-m
      (-> (t) (effect IO) Unit))
  (unregister-roots-m
      (-> () (effect IO) t)))

(foreign-import "register-m" register-m
  (-> (RV) (effect IO) Unit))
(foreign-import "unregister-m" unregister-m
  (-> () (effect IO) RV))

(define (register0 (*))  unit)
(define (unregister0)    (*))
(define (register1 (* val))
  (register-m val))
(define (unregister1)
  (* (unregister-m)))
(define (register2 (* val1 val2))
  (register-m val1)
  (register-m val2))
(define (unregister2)
  (let* ((val2 (unregister-m))
         (val1 (unregister-m)))
    (* val1 val2)))
```

(a) Moving root registration

```
(define (bindMovGC thunk frame k)
  (register-roots-m frame)
  (let* ((val    (thunk))
         (frame2 (unregister-roots-m)))
    (k frame2 val)))

(axiom ((frm *)) ((? (MRoots frm)))
       (? (Frame frm RV RV))
  (mkFrame bindMovGC))
(axiom () () (? (MRoots (*)))
  (mkMRoots register0 unregister0))
(axiom () () (? (MRoots (* RV)))
  (mkMRoots register1 unregister1))
(axiom () () (? (MRoots (* RV RV)))
   (mkMRoots register2 unregister2))
```

(b) Instantiation

Fig. 9: Instantiation for a moving stop-and-copy GC

orthogonal to the original goal of Game, which is to separate concerns *between* and not *whitin* the evaluator and the runtime.

5 Related Work

In the context of modularity in interpreters, monads and monad transformers proved to be a very useful abstraction technique [11,17]. The notion of defunctionalized monadic style incorporates the abstraction power of monadic style. Furthermore, both approaches share the use of type classes to express the monadic operators and vary their behavior. However, the work on monadic interpreters focussed on modularization of individual language constructs, without considering the supporting runtime. Game on the other hand, specifically considers the supporting runtime, and its relation with the evaluator as a whole.

PyPy [14] is both a toolchain for implementing virtual machines in RPython (a restricted subset of Python) and a meta-circular implementation of Python (in RPython). PyPy translates virtual machines written in RPython to C, and in the process also injects required runtime functionality. PyPy supports variations of the memory management strategy: the default translation uses a conservative GC but it is also possible to choose a mark-and-sweep GC. However, in PyPy, the GC must be specified in relation to RPython, whereas Game allows the GC to be tailored towards the implemented evaluator.

The combination of transformation into continuation-passing style and defunctionalization (transforming the continuations into data structures) was introduced in [13] and extensively used to relate evaluators and abstract machines [2,4], but also to restructure programs for the Web [8]. Game builds on this

```
(foreign-import "push-rv" push-rv
  (-> (RV) (effect IO) Unit))
(foreign-import "pop-rv" pop-rv
  (-> () (effect IO) RV))

(foreign-import "pushCnt" pushCnt
  (forall ((ke !*))
    (-> ((-> (RV) ke RV))
        (effect IO)
        Unit)))
(foreign-import "popCnt" popCnt
  (-> () (effect IO) (-> (RV)
                         (effect IO)
                         RV)))
(foreign-import "stack-empty?"
                stack-empty?
  (-> () (effect IO) Bool))
```

(a) Pushing and popping values and continuation functions

```
(define (bindStack thunk frame k)
  (push frame)
  (pushCnt (lambda (val) (k (pop) val)))
  (thunk))

(define (engine val)
  (if (stack-empty?)
      val
      (engine ((popCnt) val))))

(axiom ((frm *)) ((? (Stack frm)))
       (? (Frame frm RV RV))
  (mkFrame bindStack))
(axiom () () (? (Stack (*)))
  (mkStack push0 pop0))
(axiom () () (? (Stack (* RV)))
  (mkStack push1 pop1))
(axiom () () (? (Stack (* RV RV)))
  (mkStack push2 pop2))
```

(b) Instantiation

Fig. 10: Instantiation for an explicit stack

work and automates the transformation, using a type and effect system to apply it selectively. A similar idea is also used in [12], which introduces a selective CPS transformation to implement non-local control flow constructs, and is also used to implement delimited continuations in Scala [15]. It has however not been used in the work on defunctionalized interpreters [4], where instead the transformation was applied manually. Furthermore, Game goes beyond the level of the abstract machine, by instantiating it with a supporting runtime, yielding an customized and executable interpreter.

6 Conclusions and future work

We presented Game, an environment consisting of a programming language and toolchain for constructing customized interpreters. Using Game, a reusable evaluator is converted into a generic abstract machine, which is subsequently instantiated with a runtime, giving rise to a customized interpreter. Our experiments demonstrate that the high-level SICP evaluator can be reused in three different interpreters, of which the runtime varies in memory management (a non-moving mark-and-sweep versus a moving stop-and-copy GC) and in the stack discipline (managing an explicit stack).

We are currently investigating how to apply Game to other runtime variations, such as tail-call optimization [9], first-class continuations [3], the structure of the interpreter loop [6], and support for implicit parallelization using continuators [10]. The results for GC strengthen our confidence that these concerns can also be injected in high-level evaluators.

References

1. H. Abelson, G. J. Sussman, and with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd edition, 1996.

2. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proc. of the 5th ACM SIGPLAN intl. conf. on Principles and practice of declaritive programming*, PPDP '03, pages 8–19, New York, NY, USA, 2003. ACM.
3. W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher Order Symbol. Comput.*, 12:7–45, April 1999.
4. Olivier Danvy. Defunctionalized interpreters for programming languages. In *Proc. of the 13th ACM SIGPLAN intl. conf. on Functional programming*, ICFP '08, pages 131–142, New York, NY, USA, 2008. ACM.
5. Stephan Diehl and Peter Sestoft. Abstract machines for programming language implementation. *Future Gener. Comput. Syst.*, 16:739–751, May 2000.
6. M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003.
7. Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *Proc. of the fourth ACM SIGPLAN intl. conf. on Functional programming*, ICFP '99, pages 18–27, New York, NY, USA, 1999. ACM.
8. Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In *Proc. of the 16th IEEE intl. conf. on Automated software engineering*, ASE '01, pages 211–, Washington, DC, USA, 2001. IEEE Computer Society.
9. Chris Hanson. Efficient stack allocation for tail-recursive languages. In *Proc. of the 1990 ACM conf. on LISP and functional programming*, LFP '90, pages 106–118, New York, NY, USA, 1990. ACM.
10. Charlotte Herzeel and Pascal Costanza. Dynamic parallelization of recursive code: part 1: managing control flow interactions with the continuator. In *Proc. of the ACM intl. conf. on Object oriented programming systems languages and applications*, OOPSLA '10, pages 377–396, New York, NY, USA, 2010. ACM.
11. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of the 22nd ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.
12. Lasse R. Nielsen. A selective cps transformation. *Electr. Notes Theor. Comput. Sci.*, 45:311–331, 2001.
13. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of the ACM annual conf. - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.
14. Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symp. on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.
15. Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proc. of the 14th ACM SIGPLAN intl. conf. on Functional programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM.
16. T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proc. of the 13th ACM SIGPLAN intl. conf. on Functional programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.
17. Mark Snyder, Nicolas Frisby, Garrin Kimmell, and Perry Alexander. Writing composable software with interpreterlib. In *Proc. of the 8th Intl. Conf. on Software Composition*, SC '09, pages 160–176, Berlin, Heidelberg, 2009. Springer-Verlag.
18. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111:245–296, June 1994.