Schematic: A Concurrent Object-Oriented Extension to Scheme

Kenjiro Taura and Akinori Yonezawa

University of Tokyo

Abstract. A concurrent object-oriented extension to the programming language Scheme, called Schematic, is described. Schematic supports familiar constructs often used in typical parallel programs (`future` and higher-level macros such as `plet` and `pbegin`), which are actually defined atop a very small number of fundamental primitives. In this way, Schematic achieves both the convenience for typical concurrent programming and simplicity and flexibility of the language kernel. Schematic also supports concurrent objects which exhibit more natural and intuitive behavior than the "bare" (unprotected) shared memory, and permit intra-object concurrency. Schematic will be useful for intensive parallel applications on parallel machines or networks of workstations, concurrent graphical user interface programming, distributed programming over network, and even concurrent shell programming.

1 Introduction

Programmers in the world, we believe, will begin to use *concurrent* languages for various applications including demanding and intensive computation, distributed programming over networks, user interface programming and even text file processing. Although the task of concurrent programming is, in general, more difficult than sequential programming, there are many evidences and trends that support the above prospect.

- First, parallel machines will become ubiquitous. There is a strong economical demand that parallel intensive applications should run not only on dedicated parallel machines (e.g., CM5, AP1000, T3D, and Paragon), but also on networks of workstations [5, 33]. Recent research [2] has demonstrated that, with suitable communication infrastructures, intensive applications perform well on networks of workstations. Concurrent languages provide ease of programming, portability, and efficiency of applications on such computing environments.
- Second, multiple threads and synchronizing data structure (*e.g.,* concurrent objects) supported by concurrent languages allow more natural and terse description of certain types of applications. Important applications include graphical user interface (GUI) and interactive distributed computation. For example, in interactive distributed applications such as WEB browsers, the programmer in a sequential language must write a complicated scheduler loop which polls inputs both from the user and the remote server. Such

applications can be described much more concisely if the language supports multiple threads of control within a processor. In such languages, a thread can simply block when necessary data has not yet arrived. The runtime system schedules threads and guarantees that the entire application does not block as long as there is at least one runnable thread. A similar situation arises when GUI applications wish to handle multiple inputs in parallel. Since certain types of input must be synchronized with other tasks (*e.g.,* two "redraw" requests to a window must be mutually excluded), the programmer of sequential languages must suspend/restore a thread of control explicitly. This kind of mutual exclusion can be naturally expressed by multiple threads + suitable synchronizing data structures.

In summary, concurrent languages serve as a vehicle both for driving parallel machines more easily *and* expressing certain problems more naturally.

Based on the above observation, we designed and implemented a parallel extension to the programming language Scheme, named Schematic. This paper focuses on its language design. The extension is *concurrency* and *object-orientation*—the language is based on a set of flexible primitives for concurrency and a safe means for dealing with mutable data structure. We believe that the design of Schematic interests two types of concurrent language designers. First, designers who wish to extend an already popular sequential language into parallel one will be interested in how Schematic naturally *integrates* powerful concurrency primitives into existing sequential features such as function calls. Second, those who design a new parallel language, perhaps based on a concurrent calculus, will be interested in how concurrent primitives + a set of simple syntactic tricks provide a concise and familiar syntax both for sequential and parallel constructs. They are beneficial for lowering the learning barrier of the language while keeping the simplicity of the computation model and implementations of the language.

Target applications of Schematic include intensive applications (irregular symbolic or algebraic computation, in particular[1]), interactive applications (GUI in particular), and distributed programming over networks. For irregular intensive applications, Schematic supports very efficient fine-grain thread creation and communication. We have already demonstrated runtime techniques for creating and scheduling excess parallelism within a processor with very low overhead (a local thread creation + reply value communication approximately take ten RISC instructions) [30, 32]. For GUI applications, we are currently working on a GUI library where each widget is represented as a concurrent object and multiple

[1] We are not saying that numeric programs do not benefit from languages like Schematic. In fact, it is widely known that many numerics benefit from support of irregular data structures [8, 12, 13] and this leads to many proposals of extensions to C++ [6, 7, 11, 19]. The reason why we did not include numerics from the main target applications is just that in our initial implementation, floating point numbers have boxed (hence slow) representation for the simplicity. We are also working on a similar, but statically typed language called ABCL/*f* [31], which focuses on the performance of irregular numerics in fine-grain concurrent object-based languages.

events are delivered simultaneously. Since method invocations on a widget is arbitrated by the runtime system, almost no further complication is added from the programmer's point of view, while processing multiple events in parallel.

The rest of the paper is organized as follows. After giving a brief overview of Schematic in Sect. 2 and some background in Sect. 3, we introduce the basic concurrency primitives and the concurrent object-oriented extension in Sect. 4 and Sect. 5, respectively. Section 6 demonstrates some examples which highlight the main features of Schematic. Sect. 7 compares Schematic to a wide range of related languages. We finally conclude and summarize the current status of the Schematic project in Sect. 8.

2 Schematic Overview

The following is the summary of the key extensions made to Scheme:

Channels: As the fundamental primitive for synchronization, we provide first-class *channels*. A channel is a data structure on which synchronized read/write can be performed. Channels can be passed to other processes or stored in any data structure.

Future: As the fundamental construct for expressing parallelism, we introduce a variant of the *future* construct originally proposed by Halstead [14]. The value of a future expression is a channel, which we call *reply channel* of the invocation. The result of an invocation can be extracted from the reply channel of the invocation.

Explicit Reply: The reply channel of an invocation is visible from the invoked process and subject to any first-class manipulation. For example, an invocation can delegate the reply channel to another invocation, or can store the reply channel into a data structure. These features allow us to express many flexible communication/synchronization patterns in a natural way.

Concurrent Objects: Concurrent objects are supported as a safe and convenient way for sharing *mutable* data structures among concurrent processes. A concurrent object is a data structure where a method invocation can be regarded as an *instantaneous* mutation on that object. That is, the programmer is free from the complexity which comes from interleaving execution of multiple methods. An object behaves *as if* methods were serialized.

Concurrent Accesses: While achieving the instantaneous property of a method invocation, we still allow a certain amount of concurrency between multiple method invocations on a single concurrent object. In particular, we guarantee that read-only methods are never blocked by other (possibly writing) methods.

3 Background

This section briefly surveys related work which directly influenced the design of Schematic. A thorough comparison to other concurrent languages is given in Sect. 7.

3.1 Concurrent Calculi

Concurrent calculi, such as HACL [20] and π-calculus [21], have been drawing much attention and some languages have been designed based on them [23, 24]. The goal is to identify the 'core' language which expresses various computation patterns by a small number of fundamental primitives. In their simplest term, both HACL and π-calculus are based on channels communicating via processes. Channels are first-class citizens which can be passed to other processes, sent through other channels, and stored into data structures. Processes can communicate values by synchronized read/write primitives on shared channels.

Although these concurrent calculi are simple and powerful, expressing everything in the pure calculi is tedious. For example, a sequential function call would be expressed by two processes (the caller and the callee) communicating the result value via a channel. Thus, the practical concern when designing a language based on them is how to incorporate familiar constructs (*e.g.,* sequential/parallel function calls) into the language, while keeping the purity of the core.

The design of Schematic achieves both the simplicity of the core and familiar/convenient syntax for frequently used idioms such as `future` calls. A `future` call, for example, is understood as a combination of a channel creation and a process invocation. Even higher-level constructs are realized using channels and/or futures (and are defined as macros, as in Scheme).

The semantics of Schematic can be understood by encoding it into an untyped subset of HACL. Our optimizing compiler which is currently under development uses this untyped subset of HACL as the intermediate language and we are now investigating the analysis and optimization on the simple intermediate language.

3.2 Linearizable Objects

Herlihy et.al [17] defined "linearizability," which captures and formalizes an intuitively correct behavior of data structure shared by concurrent processes. An execution of a program consists of a sequence of events (history), each of which is either an invocation or a termination of a method invocation. A history is *linearizable* if events can be reordered, preserving the order of methods[2] in the original history, to a sequential history, a history in which method executions do not interleave. By definition, a method invocation in a linearizable history appears to take effect *instantaneously*. This simplifies reasoning about behavior of concurrent data structure.

Almost all concurrent object-oriented languages guarantee linearizable histories. Traditionally, many of them guarantee linearizability by, implicitly or explicitly, mutually excluding (serializing) method invocations on a single object.

As demonstrated in a separate paper by Herlihy [16], guaranteeing linearizable history, *per se,* does not require mutual exclusion. We adopt a similar implementation technique to achieve linearizability while permitting certain amount

[2] We say method M_1 proceeds method M_2 if the termination of M_1 proceeds the invocation of M_2.

of concurrent accesses to a single object. In Schematic, methods which do not update an object require no mutual exclusion, thus never be blocked by other methods. Methods which do update may still be blocked by other updating methods, but our scheduler never retries interrupted computation. The resulting scheduler is less permissive than Herlihy's with respect to deadlock, but will be more efficient because their implementation requires extra memory store due to the provision for possible retries.

4 Basic Parallelism and Synchronization Primitives

One of the underlying principles of the design of Schematic is the view that *a function/method invocation is, whether it is sequential or asynchronous, just a special case of a process creation.* More precisely, when we have some way for process creation and communication between processes, and we regard a Scheme lambda expression as (a template of) processes, a function call is achieved by invoking a thread which will put the result value to a communication medium. A sequential call just tries to get the result value immediately, while an asynchronous call at a later time.

In Schematic, both processes and the medium for inter-process communication, which we call *channel*, are first-class entities, just as functions are first-class in Scheme. This guarantees the flexibility of Schematic in the sense that whatever can be expressed in HACL or π-calculus has an obvious counterpart in Schematic.[3] This is true to other languages which support first-class channels and processes [24, 25]. However, Schematic better *integrates* parallel extensions with the sequential part and more concisely expresses frequent parallel programming idioms than those languages.

4.1 Channels

Channels are the fundamental entities which realize synchronization and communication between processes. Channels are *implicitly* created as the result of a process creation (see Sect. 4.3), or can be explicitly created via the following form:

```
(make-channel).
```

Let c be a channel. We can perform following operations on c:

- `(touch` c`)`—extracts a value from c. The value is supplied to the enclosing expression.

[3] This is not strictly true for π-calculus because writing a value to a channel in Schematic is asynchronous, while it is *synchronous* in π-calculus, in the sense that writing to a channel in a π-calculus specifies a post action which is executed *after* the reply has been completed. We presume this rarely makes difference in practice, and a synchronous call can be emulated by composing asynchronous ones, although it is tedious.

– `(reply` x c`)`—puts x in c. The enclosing expression immediately gets an unspecified value.

There may be multiple pairs of `touch/reply` performed on a single channel. In such cases, the extracted value is an arbitrary one which has been put until that time.

4.2 Process Templates (or Lambda)

A *process template* in Schematic is expressed by a lambda expression. As its syntax indicates, it is the analogue of a function in Scheme, but is given a name "process template" because applying values to it invokes a new concurrent process. Details about process invocations is described in Sect. 4.3 and this subsection concentrates on process templates.

The canonical form of process template has the following syntax:

`(lambda (`*args* $\cdots$`) (:reply-to` r`)` *exprs* $\cdots$`).`

In addition to the list of parameter names, (*i.e.,* `(`*args* $\cdots$`)`), a process templates takes another parameter, which we call *reply channel*. In the above, the name of a reply channel is specified as r.

For example, expression

```
(lambda (x) (:reply-to r)
  (reply (+ x 1) r))
```

represents a template of processes which reply `x + 1` to the given reply channel.

A reply channel can be manipulated as first class data. In particular, a process can store it into any data structure to reply a value later. For example,

```
(lambda (x) (:reply-to r)
  (set! g r))
```

expresses a template of processes which assign the given reply channel to `g` and do not reply any value to `r` from within the processes.

This is an upper-compatible extension of Scheme in the following sense. If a lambda expression does not specify `(:reply-to` r`)` clause, it is interpreted as an abbreviation of a template of processes which reply the last evaluated value to the given reply channel. That is,

`(lambda (x)` *exprs*`)`

is an abbreviation of

`(lambda (x) (:reply-to` r`) (reply (begin` *exprs*`)` r`)),`

where r is a name which does not occur in *exprs*.

In essence, we add an extra parameter to each lambda expression, the parameter which represents the location where the result value should be stored. Explicit reply gives the programmer the ability to *decouple* the termination of

a process and the delivering the result value; a process may reply a value earlier than its termination and continue some computation, reply values multiple times, or defer the reply until some synchronization/resource constraints are satisfied.

4.3 Process Invocation (or Future)

Suppose f be a process template (lambda expression). The canonical form of process creation is

(future (f *args* $\cdots$) :reply-to r).

This expression creates a new thread of control which executes the body of f with given arguments and the reply channel r. The entire expression returns r.

For example, when f is defined as

```
(lambda (x) (:reply-to r) (reply (+ x 1) r)),
```

code fragment

```
(let ((r (make-channel)))
  (future (f 3) :reply-to r)
  (future (f 4) :reply-to r)
  (touch r))
```

evaluates to 4 or 5, depending on which process replies the value to `r` first.

There are several syntactic rules which make expressions in frequent cases more concise. First, when `:reply-to` clause is omitted, a newly created channel is supplied. That is,

(future (f *args*$\cdots$))
$\equiv$ (future (f *args*$\cdots$) :reply-to (make-channel)).

Second, a function call expression found in Scheme abbreviates an expression which `touch`es the reply channel immediately after a `future` call. That is,

(f *args* $\cdots$) $\equiv$ (touch (future (f *args* $\cdots$))).

This complements the syntax rule about the abbreviation of explicit reply channel name described in the previous subsection. That is, when f is a lambda expression without explicit reply channel name, (f *args* $\cdots$) can be understood just as sequential function call in Scheme.

4.4 Higher-Level Constructs

In addition to the basic primitives, we provide several useful high-level constructs. These are:

plet: a parallel version of `let`, which evaluates all bound values in parallel.
pcall: a parallel version of apply (evaluates all arguments in parallel)

pbegin: a parallel version of `begin`, which evaluates all subexpressions in parallel.

pmap: a parallel version of `map`, which applies a given function to all elements of the list in parallel.

pfor-each: a parallel version of `for-each`.

They are defined as simple processes and/or macros.

5 Concurrent Object-Oriented Extension

Schematic extends Scheme with concurrent objects which serve as a stylized means for safely using mutable data structure in concurrent applications. Scheme does have mutable data structures (cons cells, strings, vectors, symbols are all mutable), but they are not enough for concurrent applications; interleaving executions of multiple transactions on a single data may result in a state which were impossible in non-interleaving ones. This significantly complicates the behavior of shared data and becomes the source of irreproducible bugs.

A concurrent object in Schematic exhibits simpler and more intuitive behavior than the 'bare' shared memory. The most important property is the *instantaneousness* of a method invocation: from the programmer's point of view, a method invocation appears to mutate the state of the object at some *point* between its invocation and termination. Hence the programmer never has to reason about how potential interleaving executions of concurrent method invocations[4] may affect the result. An object behaves *as if* method invocations on the object are serialized.

This behavior, however, should not be confused with an *implementation* strategy, which really serializes all method invocations on a single object. Such implementation not only loses concurrency, but also significantly narrows the range of deadlock-free programs, enforcing unnatural description of many algorithms. The problem has been recognized for a long time and in fact many languages provide some solutions to the problem. Until recently, few of them guarantees the instantaneousness of a method invocation when the programmer specifies not to serialize certain methods [7, 9, 35]. In such languages, it was up to the programmer that guarantees the desired result on all possible interleavings.

More recent languages such as SYMPAL [3] and UFO [26, 27] propose more complete solutions. They allow concurrent accesses to a single object, while guaranteeing the instantaneousness. The basic idea is to start subsequent invocations as soon as the last update on the object is done in the current method. The remaining issue is how to detect the point where the last update is done. In UFO, updates are specified by individual assignments on instance variables and the

[4] Here, we say method invocations M and M' are concurrent if there are no data dependencies that guarantee they never overlap. Notice that this definition is independent of any implementation or scheduling strategy that determines if they are really scheduled in parallel.

compiler approximates the point of the last update. In SYMPAL, a special syntax called *finally* is introduced. A finally expression performs all (hence the last) updates in a method at once and continues other computation in parallel with subsequent method invocation(s). We adopted finally construct in Schematic and add a further extension to allow read-only methods to proceed without any lock.

5.1 Classes and Methods

Defining Classes. A class is defined by `define-class` and a method either by `define-method` or `define-method!`. For example,

```
(define-class point ()
  x
  y)
```

defines class called `point`, each instance of which has slots called `x` and `y`. What follows after the class name is the list of inherited classes. For example,

```
(define-class color-point (point)
  color)
```

defines `color-point` class, each instance of which now has slot `color` in addition to `x` and `y`.

A `define-class` implicitly defines a function with the class name which creates an instance of the class. For example, an instance of `point` class is created by:

```
(point 2.0 3.0).
```

Defining Methods. The following defines a method which returns the distance between the point and the origin.

```
(define-method point (distance self)
  (sqrt (+ (* x x) (* y y)))).
```

`Define-method` defines a process template which can read instance variables of the first parameter (`self` in this example). Invoking methods has exactly the same syntax as invoking normal process templates. For example, `distance` method can be called by:

`(distance` p`)` or,

`(future (distance` p`)),`

where p is an instance of `point` (or one of its subclasses).

Explicit reply channels can be used in methods as well. For example, `distance` method equivalent to the above one could be written by:

```
(define-method point (distance self)
  (:reply-to r)
  (reply (sqrt (+ (* x x) (* y y))) r)).
```

Updating States. Updating the state of an object is expressed by `become` construct which specifies new values for updated slots as well as the result of the entire expression. Our `become` is different from that of Actors [1] in that ours specifies the result value and only allows changing state variables.[5] This construct was originally proposed by Aridor [3] in the name of `finally` construct.

For example, the following method increments `x` and `y` by `dx` and `dy` respectively, and returns the value of `(redraw! self)`.

```
(define-method! point (move! self dx dy)
  (become (redraw! self) :x (+ x dx) :y (+ y dy)))
```

The first argument of a `become` (`(redraw! self)` in this case) is called *result expression* of the `become` and specifies which value the `become` is evaluated to, while the rest part the updated values for instance variables.

There are two syntactic rules about the position of `become`s. First, a `become` can only appear in the body of `define-method!` and cannot appear in the body of `define-method`. Second, inside the body of a `define-method!`, a `become` can appear only at *tail* position of the method body. A tail position of a method body is a position where a tail function call can be put. For example, we permit

```
(define-method! class (method self ...)
  (if ...
      (become ...)
      (become ...)))
```

and,

```
(define-method! class (method self ...)
  (let ((x ...))
    ...
    (become ...))),
```

because these `become`s are, if replaced by a function call, tail calls. On the other hand, we reject

```
(define-method! class (method self ...)
  (+ (become ...) 10)).
```

By the second restriction, we guarantee that `become` is performed at most once in a method invocation. Precise definition of the syntactic restriction is not given here. It defines right places for each essential syntax and builtin Scheme macros (such as `do`).

5.2 Concurrency Semantics

Concurrency semantics refers to the way in which the programmer reasons about deadlock and liveness. Notice that it does not tell the programmer which pair

[5] `Become` in Actor allows us to replace the class of the object.

of methods are really scheduled in parallel. It merely tells which programs are guaranteed to run without deadlock.

In many concurrent object-oriented languages [4, 36], method invocations on a single object are serialized. In other words, the system schedules methods so that any pair of methods on a single object does not interleave. This is a very naive way to guaranteeing the instantaneousness of a method invocation and enforces unnatural coding styles just for avoiding possible deadlocks.

To illustrate the problem, consider a possible description of a relaxation step on a one dimensional mesh.

```
(define-class cell ()
  value          ; the value in the current step
  new-value      ; the value in the next step
  left           ; cell object on the left
  right)         ; cell object on the right

(define-method cell (current-value self)
  value)

(define-method! cell (relax! self)
  (let ((lv (current-value left)) (rv (current-value right)))
    (become #t :new-value (/ (- (+ lv rv) (* 2 value)) 2))))
```

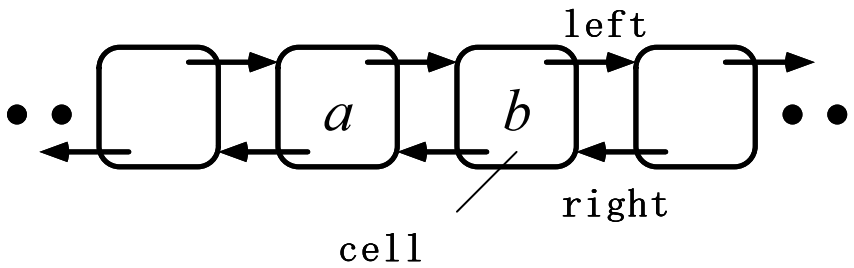

Fig. 1. Cell objects linked by `left` and `right` fields.

Many cell objects form a doubly linked list via `left` and `right`, as in Fig. 1. A relaxation step invokes `relax!` method on all the cell objects. Each `relax!` method first asks the current value of its neighbors by `current-value` method and updates itself using these values. If any pair of methods on a single object cannot overlap, invoking `relax!`s in parallel may result in deadlock. This happens when two neighboring objects start their `relax!` method almost simultaneously. Invocation of `relax!`es never terminate unless `current-value`s invoked from within them terminate, but these `current-value`s in this scheduling wait for the completion of `relax!`es!.

Notice that since instance variable `value` is not updated in `relax!`, there is no reason why we serialize `current-value` and `relax!` on a single object. This example suggests that we must have a finer classification of methods, which defines which types of methods can/cannot interleave with which.

The next example demonstrates another requirement for the concurrency between methods. Consider a sorted (linear) list of concurrent objects and a method which inserts a new value in the appropriate place, maintaining the list to be sorted.

```
(define-class cell ()
  value
  next)

(define-method cell (get-value self)
  value)

;;; Insert V in the appropriate place
(define-method! cell (insert! self v)
  (if (< v (get-value next))
      ;; V is smaller than VALUE of NEXT,
      ;; so we insert V between SELF and NEXT
      (become 'done :value value :next (cell v next))
      ;; otherwise recurse on the child
      (become (insert! next v))))
```

We easily see that, if we serialize methods to a *single* object, accesses to an entire list is also serialized, because `insert!` on the head object finishes only when the entire computation finishes. Since the "else" branch of the above method simply delegates the `insert!` method to `next` object, we can accept subsequent methods as soon as `(< v (get-value next))` turns out to be false. This example suggests that we must provide a way to accept subsequent methods when a method execution reaches a certain point.

Based on the above observation, Schematic refines the traditional mutual exclusion model in the following two ways.

- We classify methods into two types, *i.e.,* those defined by `define-method` (which we call `method` below) and those defined by `define-method!` (which we call `method!` below). Schematic guarantees that a `method` always progresses and can overlap with any other `methods` and `method!`s.
- For solving the second example, we breakdown execution of a `method!` into two stages, called *before-stage* and *after-stage*. After-stage evaluates the result expression of the `become` and before-stage performs all actions before the after-stage. In the `move!` method, for example,

  ```
  (define-method! point (move! self dx dy)
    (become (redraw! self) :x (+ x dx) :y (+ y dy)))
  ```

the before-stage consists of evaluating `(+ x dx)` and `(+ y dy)` and updating the instance variables. After-stage invokes `redraw!` method for self. Our relaxed mutual exclusion rule is that a before-stage of an invocation cannot overlap with before-stages of other invocations, but an after-stage can overlap with other before-stages and after-stages. Hence, this example does not deadlock.

The first rule implies that, as far as deadlock is concerned, we only have to consider `method!`s. The above relaxation example never causes deadlock because it only invokes one `method!` per an object. The second rule says that a `method!` can release the mutex associated to an object earlier than its termination. The second example allows multiple `insert!`s to operate in parallel on a list because a cell object can accept subsequent methods as soon as it decides to call `next` object.

When the system results in a situation where no executing methods cannot be completed, the system simply deadlocks and never retries nor aborts unlike schedulers in many database systems.

5.3 Consistency Semantics

Concurrent objects exhibit simpler and more intuitive behavior than the regular shared memory, in that methods on concurrent objects interleave in the granularity of an entire method body, rather than individual load/store operations. This section gives the description of how to reason about possible states (*i.e.,* values of instance variables) of an object at any given time. There are two important rules.

- First, values of instance variables never change inside a body of both types of methods (*i.e.,* `define-method` and `define-method!`); their values are fixed at the beginning of the method. A mutation by a `method!` takes effect in methods *invoked after* the before-stage of the `method!`. Consider the following counter object.

```
(define-class counter ()
  value)

(define-method counter (get-value self)
  value)

(define-method! counter (add-value! self x)
  (become value :value (+ value x))).
```

  Method `add-value!` increments `value` of a counter object by `x`. Referencing `value` in the result expression of the `become` still reads the *original* value of `value`. On the other hand, if we change `add-value!` method in the following way:

```
(define-method! counter (add-value! self x)
  (become (get-value self) :value (+ value x))),
```

we will obtain the value *after* `value` has been incremented, since `get-value` method is invoked after the update has been done.

– Second, `become` atomically updates all the instance variables, no matter how many variables are updated. Consider the following example.

```
(define-method! point (move! self dx dy)
  (become (redraw! self) :x (+ x dx) :y (+ y dy)))

(define-method point (position self)
  (list x y))

(let ((p (point 0 0)))
  (pbegin
    (move! p 1 1)
    (position p))).
```

In the last expression, we invoke `position` method on a point object that is moving from point $(0, 0)$ to point $(1, 1)$. The `position` method is guaranteed to return either `(0 0)` or `(1 1)`. It never obtains `(0 1)` nor `(1 0)`.

6 Examples

6.1 Concurrent Tree Updating

This example demonstrates how the concurrency semantics of our model, that is, classification of methods and the notion of before/after-stage, allows natural description of concurrent data structure. Consider a binary tree search algorithm where each node of the binary tree is a concurrent object. Here is the definition of each node object.

```
(define-class bintree-node ()
  ;; remember association between KEY and VALUE
  key
  value
  ;; children (#f when it does not exist)
  left
  right)
```

Each node has its key and associated value. It holds that the key of the left child is less than that of self and the key of the right child is greater than that of self. Hence binary search operation is very straightforward.

```
;;;
;;; Lookup the value associated for K.
;;;
```

```
(define-method bintree-node (lookup self k not-found)
  (cond ((= k key) value)                ; found
        ((< k key)
         ;; look for the left subtree if K < KEY
         (if left
             (lookup left k not-found)
             not-found))
        (else
         ;; look for the right subtree if K > KEY
         (if right
             (lookup right k not-found)
             not-found))))
```

Since this operation does not update the tree, we use `define-method`, hence multiple `lookup` invocations can simultaneously operate on a single tree. The following method installs a new association between key `k` and value `val`.

```
;;;
;;; Establish association K ↔ VAL, maintaining the
;;; following invariant:
;;;    "(KEY of LEFT) < (KEY of SELF) < (KEY of RIGHT)"
;;;

(define-method! bintree-node (insert! self k val)
  (cond ((< k key)
         (if left
             ;; if there is already left child, delegate this value
             ;; to the child, unlocking self
             (become (insert! left k val))
             ;; if there is no left child, create it
             (become #t :left (make-leaf-bintree-node k val)))
        ((= k key)
         (format #t "Warning conflicting key (~s ~s)~%" key value)
         (become #f))
        (else
         ;; the same algorithm as the first case, but for the right child
         (if right
             (become (insert! right k val))
             (become #t :right (make-leaf-bintree-node k val)))))))
```

This method first finds the appropriate place to which we insert the item and then installs a new node to that place. An interesting case happens in internal nodes; an internal node recursively calls `insert!` method for an appropriate child *after* it unlocks `self` for subsequent requests. This is expressed by

```
(become (insert! left k val))
```

at line 6 and

```
(become (insert! right k val))
```

at line 15. As has been described in Sect. 5.2, these recursive calls are in the after-stage of the method, *i.e.,* executed after `self` has been unlocked.

6.2 Synchronizing Objects

To demonstrate the expressive power of explicit reply channels, consider implementation of an object which embodies an application-specific synchronization constraint. That is, upon a method invocation, the object defers the reply of the invocation until certain synchronization constraints are satisfied *by subsequent methods.* Simply blocking computation *inside* the method does not work, because this may exclude subsequent method invocations, thus block the original computation forever!

As a simple example, consider implementing a synchronizing stack object. The synchronization constraint is that pop operation on an empty stack should block until the next push operation has been made. An instance of the following `stack` class has two instance variables `values` and `waiters`. `Values` is a list of pushed values and `waiters` a list of reply channels of pop requests which are not yet served. At least one of `values` or `waiters` is always empty.

```
(define-class stack ()
  values ; list of pushed values
  waiters) ; list of reply channels
```

The following `stack-pop!` method facilitates explicit reply channel feature of Schematic. The method first checks if `values` is empty. If it is, we block the caller by not replying any value to the reply channel of the invocation. In order to later unblock the caller, we insert the reply channel to `waiters` list. Otherwise we simply serve the top element of `values` by `reply` operation on the reply channel.

```
;;;
;;; Pop a value from the stack.  Block if empty, until the next STACK-PUSH!
;;;

(define-method! stack (stack-pop! self)
  (:reply-to r)   ; declare the name of the reply channel to be R
  (if (null? values)
      ;; Stack is empty.  Does not reply anything and let the caller
      ;; wait until the next value comes
      (become #t :values '() :waiters (cons r waiters))
      ;; Stack is not empty.  Simply reply the top element to R.
      (become (reply (car values) r)
      :values (cdr values) :waiters '())))
```

To make this example complete, we give the description of push operation below. The method first checks `waiters` list. If it is empty, we simply push the value to `values` for servicing later `stack-pop!`s. Otherwise it removes a channel from `waiters` and serves the value to it.

```
;;;
;;; Push VAL on the stack.
;;;

(define-method! stack (stack-push! self val)
  (if (null? waiters)
      ;; nobody is waiting, hence simply push VAL
      (become #t :values (cons val values) :waiters '())
      ;; somebody is blocking, hence resume the first guy
      (become (reply val (car waiters))
              :values '() :waiters (cdr waiters))))
```

7 Comparison to Other Languages

Schematic can be related to several groups of other concurrent languages. First, Schematic is a language whose computation model is based on a concurrent calculus which gives us the foundations of compiler optimizations. Second, Schematic supports concurrent objects which allow/guarantee more concurrency than the traditional mutual exclusion model which serialize all method invocations on a single object. Third, Schematic is an extension of a popular sequential language, which already has a philosophy to be preserved.

7.1 Languages Based on Concurrent Calculi

PICT. PICT [24] is a concurrent language based on π-calculus [21]. Its design goal is to support frequently used higher-level idioms as syntactic rules in a language directly based on π-calculus (just as Scheme is based on λ calculus and has higher-level idioms such as `do` loop). Although the language design is still evolving, there seems to be no constructs which directly support future or even sequential function calls. Schematic shares the same design goal and demonstrates that, by looking at function calls and lambda expressions of Scheme in a slightly different way, a language with a very small number of fundamental primitives can at the same time provide convenient constructs (such as `future` and `plet`) for typical cases.

7.2 Concurrent Extensions to Sequential Languages

Extending a sequential language to yield a concurrent dialect has many practical advantages. Among others, Multilisp and Concurrent ML are closely related to Schematic, in that Multilisp extends Scheme by future and Concurrent ML supports first class channels.

Multilisp. Multilisp [14] is the language which originally embodies the `future` construct. The central idea of `future` is that a future expression returns something which later becomes the result value. This construct or variants are later adopted not only in parallel Lisps but also in some concurrent object-oriented languages [18, 31, 34, 36].

Schematic also supports a variant of `future`. An apparent difference between the `future` in Multilisp and the one in Schematic is that in Multilisp, producer-consumer synchronization of a future invocation is implicit in value reference, whereas Schematic requires explicit `touch` operations. For example, invoking `(f x)` and `(g y z)` in parallel and then adds the two results is written by

```
(+ (future (f x)) (future (g y z))),
```

in Multilisp, while it is written by

```
(let ((l (future (f x))) (r (future (g y z))))
  (+ (touch l) (touch r)))
```

in Schematic.[6]

Informally, the Multilisp view of a future is that what is immediately returned by a future expression is a placeholder object, which later *becomes* the result value for itself, whereas the Schematic view is that a future expression returns a placeholder (*i.e.,* channel) *into which the result value is stored.*

There are tradeoffs between these two views. The implicit future in Multilisp, as the above example indicates, often results in a terse expression but loses some flexibility. On the other hand, by making `touch` explicit, we can distinguish a reference to a channel itself from the reference to the value which is stored in the channel by the program text. This not only guarantees fast value reference without additional compiler analysis [29], but also produces more expressive power by making channels first-class citizens. Examples have been given in Sect. 6.2.

Another difference is their treatment of shared mutable data. Multilisp provides Scheme builtin data as the basis for mutable data and some atomic memory operations such as `replace-if-eq` (analogue of *compare & swap*). No higher-level mechanisms for defining safe mutable data are provided. Schematic supports and encourages the use of concurrent objects to represent mutable data, concurrent accesses to which are arbitrated by the runtime system.

Concurrent ML. Concurrent ML (CML) [25] extends SML by first-class channels and fork (`spawn`), whereas Schematic extends Scheme by first-class channels, fork (`future`), and concurrent objects. To put concurrent objects aside, the main difference is that CML does not support any higher-level concurrent primitives (parallel calls or even `futures`).

Consider how to do two CML function calls `f x` and `g x` in parallel. Since the results must now be extracted from a channel, let us define a 'wrapper' function which takes a channel and sends the result of `f x` to the channel.

[6] As far as this particular example is concerned, `pcall` would express it more nicely.

```
fun wrapper f x c = send (f x, c)
```

What remains is to create two channels, spawn two `wrapper`s, and wait for the result.

```
let c0 = channel ()
and c1 = channel ()
in
   (spawn (fn () => wrapper f x c0);
    spawn (fn () => wrapper g x c1);
    accept c0; accept c1)
end
```

Presumably, a fragment like this will appear very often and should be more stylized, as in Schematic. In fact, a restricted version of future can be defined in CML by

```
fun future f x =
  let c = channel ()
  in
    (spawn (fn () => send (c, f x)); c)
  end.
```

Except that it can only invoke a unary function, the above future takes any function and any argument and returns the reply channel. This is more monolithic and less flexible than `future`s in Schematic, in that a future now always creates a reply channel and the caller loses the chance to specify a reply channel.

Given that a function is the fundamental building block of CML programs, CML should support and encourage a convenient way for invoking functions in parallel. Schematic is designed based on this principle, while leaving chances to construct customized communication structure whenever desired.

7.3 Concurrent Object-Oriented Languages

A *concurrent object* refers to data that embodies some access arbitration mechanisms so that an execution of a method never observes inconsistent state of an object. Several object models have been proposed and they differ in the degree of concurrency on a single object. Below we compare Schematic with other languages in this respect.

Early Concurrent Object-Oriented Languages based on Actors. Some early concurrent object-oriented languages such as ABCL/1 [35, 36] and Cantor [4] achieves the instantaneousness of a method execution by mutually excluding all the method invocations on an object. This is often explained by "an autonomous object which has its own thread and message queue." Although the traditional mutual exclusion model provides the instantaneousness and a very simple model in which the programmer reasons about deadlock, it is often

criticized to serialize too much. This not only loses performance gain which is otherwise possible by exploiting parallelism, but also enforces unnatural description of algorithms to solely avoid potential deadlock.

Concurrent Aggregates. Concurrent Aggregates (CA) [9, 10] supports *aggregates* in addition to regular objects. A regular object is a serializing data structure and an aggregate is internally composed of multiple objects, but externally viewed as if it were a single object. By processing multiple method invocations on an aggregate by multiple internal objects, an aggregate can serve as a non-serializing object. Maintaining the consistency among multiple internal objects, if required, is the responsibility of the programmer.

SYMPAL and UFO An object in more recent languages such as SYMPAL [3] and UFO [26, 27] allows a running method to overlap with subsequent methods, while achieving the instantaneousness of method invocations. The basic idea is to schedule subsequent methods on an object as soon as the current method finishes the last update in the method body.

`Become` of Schematic was originally proposed in SYMPAL as `finally` construct. A method can perform at most one `finally`, which all at once updates instance variable. The syntactic rule described in Sect. 5.1 (also described in [3]) guarantees the single update rule. Schematic extends the object model of SYMPAL by further classifying methods into two types (`define-method` and `define-method!`) and guaranteeing that `define-method` always progresses without any mutual exclusion.

UFO also enforces the single assignment rule and takes another approach for detecting the last update. A method (called procedure in UFO) updates state variables by individual assignments. The compiler statically approximates the point where the last update is done and unlocks the object at that point.

C++ Dialects. Several C++ dialects support *objectwise* concurrency control mechanisms. Here we concentrate on dialects which support this type of object model and do not discuss other types of C++ extensions such as data-parallel extensions [6].

CC++ [7] does not directly support concurrent objects, but the similar effect can be achieved by `atomic` member functions. By declaring a member function as `atomic`, the member function locks/unlocks the object at invocation/termination as in the traditional Actors. Thus the object model of CC++ has the same problems with early concurrent object-oriented languages. Non-atomic functions can run concurrently with others, but this merely leaves consistency issues for the programmer.

Objects in ICC++ [11] allows two methods M and M' to operate on a single object in parallel if there are no read/write nor write/write conflicts between them *on any instance variable of the object.* In this way, ICC++ guarantees that any method appears to take effect instantaneously, while achieving concurrent accesses to a single object. The main difference between ICC++ and the

UFO/SYMPAL/Schematic group is that the ICC++ model performs mutual exclusion on a per instance variable basis, rather than a per object basis.

The range of programs which are guaranteed to be scheduled without deadlock do not seem quite different between ICC++ and Schematic. A foreseeable problem with the ICC++ object model is that each object now potentially has to have multiple locks to serialize only conflicting methods. The worst case requires a lock per instance variable and removing redundant locks requires global information on the source code.

8 Summary and Current Status

The design of Schematic, a concurrent object-oriented extension to Scheme, has been presented. Just as most part of Scheme can be understood in terms of a very simple calculus (the λ-calculus), most part of Schematic can be understood in terms of a simple *concurrent* calculus (HACL). To make it really practical, Schematic also supports and encourages the use of familiar paradigms (*i.e.,* `futures` and concurrent objects) as well, achieving both the simple core of the language and the conciseness/convenience in typical programs.

A prototype on top of a sequential Scheme (`Scheme->C`) has been implemented and is running on AP1000 and AP1000+ massively parallel processors [15, 28]. We had developed an RNA secondary structure prediction algorithm [22],[7] which is essentially a parallel tree search with application-specific priority and a load-balancing control scheme, and Barnes-Hut Nbody algorithm. Experiments on an AP1000+ system (SuperSparc 50 Mhz $\times$ 256) indicated an usable performance, though many more improvements are necessary.

Further information is available via:

`http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.

References

1. Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.
2. Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.
3. Yariv Aridor. *An Efficient Software Environment for Implicit Parallel Programming with a Multi-Paradigm Language.* PhD thesis, the Senate of Tel-Aviv University, 1995.
4. W. C. Athas and C. L. Seitz. Cantor user report version 2.0. Technical report, Computer Science Department, California Institute of Technology, 1987.
5. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] This paper describes algorithms and results by message passing C on CM5, and we are now preparing the result in Schematic.

6. F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing*, pages 588–597, 1993.
7. K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. The MIT Press, 1993.
8. Andrew Chien, M. Straka, Julian Dolby, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. A case study in irregular parallel programming. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.
9. Andrew A. Chien. *Concurrent Aggregates (CA)*. PhD thesis, MIT, 1991.
10. Andrew A. Chien and William J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Princeples & Practice of Parallel Programming*, pages 187–196, Seattle, Washington, March 1990.
11. Andrew. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the Second International Symposium on Object Technologies for Advanced Software (To appear)*, 1996.
12. High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, 1994.
13. Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing '94*, pages 439–448, 1994.
14. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, April 1985.
15. Kenichi Hayashi, Tunehisa Doi, Takeshi Horie, Yoichi Koyanagi, Osamu Shiraki, Nobutaka Imamura, Toshiyuki Shimizu, Hiroaki Ishihata, and Tatsuya Shindo. AP1000+: Architectural support of put/get interface for parallelizing compiler. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pages 196–207, 1994.
16. Maurice P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
17. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
18. Waldemar Horwat, Andrew A. Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, Portland, Oregon, July 1989.
19. Yutaka Ishikawa. The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1. Technical Report TR–94014, RWC, June 1994. `http://www.rwcp.or.jp/people/mpslab/ mpc++/mpc++.html`.
20. Naoki Kobayashi and Akinori Yonezawa. Higher-order concurrent linear logic programming. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer Verlag, 1994. `http:// web.yl.is.s.u-tokyo.ac.jp/pl/hacl.html`.
21. Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

22. Akihiro Nakaya, Kenji Yamamoto, and Akinori Yonezawa. RNA secondary structure prediction using highly parallel computers. *Comput. Applic. Biosci. (CABIOS) (to appear)*, 11, 1995.
23. Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer Verlag, 1994.
24. Benjamin C. Pierce and David N. Turner. PICT: A programming language based on the Pi-Calculus. Technical report in preparation; available electronically, 1995.
25. John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.
26. John Sargeant. United functions and objects: An overview. Technical report, Department of Computer Science, University of Manchester, 1993.
27. John Sargeant. Uniting functional and object-oriented programming. In Shojiro Nishio and Akinori Yonezawa, editors, *Proceedings of First JSSST International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1993.
28. Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, pages 288–297, 1992.
29. Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–87. The MIT Press, 1991.
30. Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 218–228, 1993. `http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html`.
31. Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/*f*: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292. American Mathematical Society, 1994. `http://web.yl.is.s.u-tokyo.ac.jp/pl/ schematic.html`.
32. Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming (TPPP)*, number 907 in Lecture Notes in Computer Science, pages 121–136. Springer Verlag, 1994. `http://web.yl.is.s.u-tokyo.ac.jp/pl/ schematic.html`.
33. Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, February 1995.
34. William Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.
35. Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System—Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

36. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Conference Proceedings*, pages 258–268, 1986.

This article was processed using the LaTeX macro package with LLNCS style