Concurrency in the Lisp Package

A. Motzek

Abstract

The Lisp Package is a Lisp interpreter / compiler written in Java. It can be embedded into Java applications when scripting is needed [1]. The Lisp Package implements a Lisp-1 [2] but otherwise tries to conform to Common Lisp, for example regarding the names of predefined functions and macros. Other features are borrowed from programming languages like Ruby, Go or Occam or were found in the literature. This paper describes the concurrency features of the Lisp Package: implicit parallelism, explicit lightweight threads, locks and channels.

1. Introduction

Contemporary processors for the consumer market (e.g. Intel Core i) have 4 cores and can handle up to 8 concurrent threads in hardware. Processors for enterprise or scientific use support several hundred hardware threads (e.g. Oracle SPARC T3 [3], Intel Xeon Phi [4]).

From these numbers it is obvious that even now single threaded software can use only a small fraction of the hardware of modern processors. And there are expectations that the trend towards more cores and threads will continue [5].

Coding explicitly parallel software has an overhead, because statements for creation and synchronization of threads or tasks must be inserted into the program by hand [6]. Furthermore it is error-prone due to the possibilities of deadlock, live lock and race conditions [7]. Favorable is therefore a programming language that parallelizes a given program automatically.

2. Implicit Parallelism

2.1 Concept

In an imperative programming language, operations with side effects make it difficult to analyze which parts of a program can be executed in parallel. Functional programming languages are better starting points for automatic program parallelization because their style favors the absence of side effects. In an application of function *F* to arguments *A1*, *A2*, and so on, in Lisp syntax

(*F A1 A2* ...)

the arguments can be evaluated in any order and even concurrently if these evaluations have no side effects. The result of the application of *F* to the evaluated arguments will always be the same.

The Lisp Package does not check if evaluating arguments has side effects. It postulates that the order of argument evaluation is not specified and that these might even be evaluated concurrently.

Sequential execution of expressions must be explicitly stated with **prog1**, **prog2** or **progn** (→ Common Lisp) or is given by the semantics of special functions as it is the case with **if**, **cond**, **and**, **or**, **when**, **unless**, **dotimes** (→ Common Lisp), **aif**, **aand** (→ anaphoric macros suggested by Paul Graham) and **ensure** (→ ensure keyword in Ruby [9], similar to unwind-protect in Common Lisp).

If within a **catch** (→ Common Lisp) expression multiple exceptions are thrown because of implicit or explicit parallelism, then catch (and catch-and-apply, see below) inspects only one exception. Other exceptions are silently discarded. It is not specified how the inspected exception is selected. If the symbols provided as first arguments to catch and **throw** (→ Common Lisp) are the same, then catch returns the thrown value. Otherwise catch lets the exception escape.

Catch is a macro that is implemented with a lower level special function catch-and-apply:

```
(setq catch
  (let
    ((get-value (lambda (name value) value)))
    (mlambda args
      (list
        catch-and-apply
        (first args)
        get-value
        (second args)))))
```

The special function **catch-and-apply** (→ inspired by the try-catch mechanism in Java) takes three arguments: first a symbol that is matched with the thrown symbol, second a function with two parameters that is invoked with the arguments passed to throw if the symbols match and third the expression to evaluate.

2.2 Implementation

Evaluation of expressions in the Lisp Package internally uses runnables, continuations, worker threads, dispatch stacks and a dispatch queue for concurrent execution.

A **runnable** (→ java.lang.Runnable) is something that performs a task when it is executed by a worker thread. The result of the task is consumed by a continuation. Therefore all runnables in the Lisp Package hold a reference to a continuation.

A **continuation** represents "the rest of the program" – and especially what is executed next. Most continuations produce more runnables. If a list of the kind (*F A1 A2* ...) is evaluated, two kinds of continuations are relevant for concurrency.

- Apply Value: This continuation receives the value of *F*, sets up a continuation of class Receive Argument for every argument, puts them into a runnables that evaluate the arguments and pushes the runnables on the dispatch stack.

- Receive Argument: This continuation receives the value of an A_i, stores it and checks if now all values for all arguments are there. If yes, the value of *F* is applied to the values of the A_i. The implementation of this continuation is lock free. It uses an atomic decrement operation on an integer (→ java.util.concurrent.atomic.AtomicInteger) when an argument value arrives. This operation compiles to "test and set" or "compare and swap" CPU instructions.

Each **worker thread** has a private, local dispatch stack. New runnables that are created within the worker thread are pushed onto its local **dispatch stack**. A worker thread first tries to pop a runnable from the local dispatch stack and only dequeues from the central dispatch queue, if the local dispatch stack is empty.

If a worker thread has an empty local dispatch stack, it announces its need for runnables. Other worker threads then take runnables from the bottom end of their dispatch stacks and enqueue them into the central dispatch queue. The effect is similar to work stealing [6], but the local dispatch stacks are not touched by foreign threads and therefore no synchronization primitives have to be used before accessing them.

Announcing the need for runnables is implemented as atomic increment operation on an integer. Worker threads that give away a runnable into the central dispatch queue use an atomic decrement operation. The integer is shared by at most 8 worker threads. If there are more worker threads, more than one integer is used.

The **dispatch queue** is a data structure that can be used by all threads concurrently. If the dispatch queue is empty, a worker thread trying to dequeue a runnable must wait until one is enqueued by another thread.

The dispatch queue uses two monitors (→ synchronized keyword in Java) to protect its read and write end from corruption by concurrent accesses and a semaphore (→ java.util.concurrent.Semaphore) which ensures that not more runnables can be dequeued than were enqueued before.

The Lisp Package sets up one worker thread for every available hardware thread in the processor(s). Worker threads are heavyweight (instances of a subclass of → java.lang.Thread).

3. Explicit Parallelism

Some algorithms are inherently sequential. Implementing and running them with the Lisp Package and its implicit parallelism shows no speed up. A simple example is counting the elements of a list:

```
(setq seq-length
  (lambda (l)
    (if
      (null? l)
      0
      (+ 1 (seq-length (rest l))))))
```

Implicit parallelism causes that 1 and (seq-length (rest l)) can be evaluated concurrently, but it is obvious that this is useless.

The Lisp Package does not use speculative execution (here: evaluating (null? l), 0 and (+ 1 (seq-length (rest l))) concurrently and choosing either 0 or the sum after the result of (null? l) becomes known) and even this would not help.

In cases like this, the function **split** can be used. The function takes two arguments, a list and a positive integer. The result of a function application is a list of lists. The input list is split into nearly equal parts.

Each of these parts can be processed by a sequential algorithm, but all parts concurrently. Then the results are combined.

For the example of counting elements of a list this leads to:

```
(setq par-length
 (letrec
   ((seq-length
     (lambda (l)
       (if
         (null? l)
         0
         (+ 1 (seq-length (rest l)))))))
    (lambda (l)
     (apply +
       (mapcar
         seq-length
         (split l *thread-count*))))))
```

The symbol ***thread-count*** is bound to an integer that gives the count of available hardware threads.

The function **unsplit** is the inverse of split. For every list *l* and positive integer *n* (unsplit (split *l n*)) is *l* again.

4. Lightweight Threads

Because of the implicit parallelism in the Lisp Package, creating threads manually is only needed in special cases. One of these special cases is a thread containing an endless loop implementing a generator (→ example in chapter 7.3.).

Threads can be created by invoking the special function **go** on an expression (→ go statement in Go [8]). This immediately returns a future. As side effect, a spawned lightweight thread starts to evaluate the expression.

"Lightweight" means that no new worker thread (→ chapter 2.2.) is created, only a new runnable is pushed on a local dispatch stack.

The future returned by the special function go can be used to wait for and then access the value of the expression. For that, the function **await-future** is applied to the future. For every expression *expr* the value of (await-future (go *expr*)) is identical to the value of *expr*.

If during the evaluation of *expr* an exception is thrown, calling await-future will throw this exception.

If await-future is invoked multiple times for the same future, the results of all invocations will be the same. The expression is evaluated only once.

5. Locks

Locks can be used to make sure that a critical section of code is not executed concurrently by more than one thread. A lock can be obtained by calling **make-lock**. This function has no parameters and returns a new lock every time it is invoked.

Protecting a critical section with a lock can be done using the following idiom:

```
(progn
  (acquire-lock l)
  (ensure
    critical section
    (release-lock l)))
```

where **acquire-lock** either proceeds immediately if no other thread executes the critical section or blocks until another thread leaves it.

A thread announces that it leaves the critical section by applying **release-lock** on the lock.

The special function **ensure** (→ ensure keyword in Ruby [9]) takes two arguments. The value of the first argument is the value of the whole ensure expression. The second argument is evaluated for its side effect even if an exception is thrown during the evaluation of the first argument.

The macro **with-lock** is an abbreviation for the idiom described above:

```
(with-lock l
  critical section)
```

6. Channels

Channels can be used for communication between concurrent parts of a program by exchanging messages. Messages are simply s-expressions.

Channels are created with the function **make-channel** (→ make for a channel type in Go). It takes one argument which has to be a non-negative integer that specifies the capacity of the channel. The capacity of a channel is the number of messages that can be send to it without blocking the sender if no messages are received from the channel.

Sending to a channel is done by using the function **send-on-channel** (→ *channel* ! *value* in Occam and *channel* <- *value* in Go). The function has three parameters: a channel, a message and a timeout in milliseconds. If the message could be send within the timeout, the message is the result of function call. Otherwise the result is nil.

The function **receive-from-channels** (→ ALT and *channel* ? *variable* in Occam, select and <- *channel* in Go) receives a message from a list of channels. The function takes two arguments. The first argument is a list of channels and the second argument is the timeout in milliseconds. If within the timeout a message can be received from one of the channels, it is returned. Otherwise the result of the function call is nil.

A channel is closed by applying **close-channel** (→ close built-in function in Go [11]) on it. Invoking send-on-channel and receive-from-channels on a closed channel immediately returns nil. For a closed channel **closed-channel?** returns t.

7. Examples

7.1. Towers of Hanoi with and without implicit parallelism

Any algorithm that uses the "divide and conquer" strategy can benefit from the implicit parallelism in the Lisp Package. An example for this is the problem of the Towers of Hanoi [12] where a problem of size *n* is divided into two problems of size *n – 1*. The straight forward translation of the algorithm into Lisp leads to:

```
(setq p-hanoi
  (lambda (n src dest help)
    (cond
      ((less? n 0) (throw (quote error) "undefined"))
      ((equal? n 0) nil)
      ((equal? n 1) (list (list src dest)))
      (t (append
        (p-hanoi (- n 1) src help dest)
        (list (list src dest))
        (p-hanoi (- n 1) help dest src))))))
```

Avoiding implicit parallelism and forcing a sequential execution requires the use of a special form like progn (→ chapter 2.1. for other sequential special forms):

```
(setq s-hanoi
  (lambda (n src dest help)
    (cond
      ((less? n 0) (throw (quote error) "undefined"))
      ((equal? n 0) nil)
      ((equal? n 1) (list (list src dest)))
      (t (let
        ((r1 nil) (r2 nil))
    (progn
      (setq r1 (s-hanoi (- n 1) src help dest))
      (setq r2 (s-hanoi (- n 1) help dest src))
      (append r1 (list (list src dest)) r2)))))))
```

The results of both functions are equal for equal arguments, but s-hanoi uses only one hardware thread.

7.2. Dining Philosophers with Locks

An example where locks can be used is the Dining Philosophers problem [13]. Five philosophers sit around a table. Each philosopher either thinks or eats. For eating she or he must first take two forks. Unfortunately, there are only five forks – one fork between each pair of philosophers sitting side by side.

Thinking is simulated by sending a log message to a channel:

```
(setq think
  (lambda (name out)
    (send-on-channel out (concatenate name " is thinking") nil)))
```

Eating takes two forks, sends a log message to a channel and waits three to six milliseconds:

```
(setq eat
  (lambda (name first-fork second-fork out)
    (with-lock first-fork
      (with-lock second-fork
        (progn
          (send-on-channel out (concatenate name " is eating") nil)
          (receive-from-channels nil (+ 3 (random 4))))))))
```

Living means thinking, then eating, then thinking again and so on:

```
(setq live
  (lambda (name first-fork second-fork out)
    (progn
      (think name out)
      (eat name first-fork second-fork out)
      (live name first-fork second-fork out))))
```

The setup of the whole scenario works by creating the channel for the log messages, creating the five forks and starting five threads for the philosophers. The channel for the log messages is returned:

```
(setq make-philosophers
  (lambda ()
    (let
      ((fork-1 (make-lock))
       (fork-2 (make-lock))
       (fork-3 (make-lock))
       (fork-4 (make-lock))
       (fork-5 (make-lock))
       (out (make-channel 1)))
      (progn
        (go (live "andi" fork-1 fork-2 out))
        (go (live "berti" fork-2 fork-3 out))
        (go (live "conni" fork-3 fork-4 out))
        (go (live "det" fork-4 fork-5 out))
        (go (live "emil" fork-1 fork-5 out)) #| not fork-5 fork-1 to avoid deadlock |#
        out))))
```

The messages that can be received from the log channel are as expected, e.g.

```
"emil is thinking" "berti is thinking" "det is thinking" "conni is thinking"
"andi is thinking" "det is eating" "det is thinking" "emil is eating"
"conni is eating" "emil is thinking" "conni is thinking" "berti is eating"
"det is eating" "berti is thinking" "det is thinking" "andi is eating"
"det is eating" "det is thinking" "andi is thinking" "conni is eating"
"emil is eating" "emil is thinking" "conni is thinking" ...
```

No more than two philosophers can eat at the same time. And if two philosophers eat at the same time, the two are not seated side by side.

7.3. Sieve of Eratosthenes using Generators and Channels

This example shows the use of generators and channels with the five functions integer-generator, next-integers, filter-multiples, sieve and prime-generator:

- integer-generator and next-integers implement a generator for integers,
- filter-multiples copies integers from an input to an output channel leaving out the multiples of a given integer,
- sieve forms a chain of filters with a filter for every prime discovered so far and
- prime-generator drives the filter chain with the integer generator.

The function integer-generator returns a channel from which consecutive integers can be received.

```
(setq next-integers
  (lambda (out current)
    (when
      (send-on-channel out current nil)
      (next-integers out (+ current 1)))))

(setq integer-generator
  (lambda (start)
    (let
      ((out (make-channel 0)))
      (progn
        (go (next-integers out start))
        out))))
```

The function filter-multiples receives integers from an input channel and passes them only to the output channel, if they are not multiples of a given factor.

```
(setq filter-multiples
  (lambda (in out factor)
    (let
      ((number (receive-from-channels (list in) nil)))
      (if
        (integer? (/ number factor))
        (filter-multiples in out factor)
        (when
          (send-on-channel out number nil)
          (filter-multiples in out factor))))))
```

The function sieve receives consecutive integers from an input channel and sends prime numbers to the output channel.

```
(setq sieve
  (lambda (in out)
    (let
      ((prime (receive-from-channels (list in) nil))
       (transfer (make-channel 0)))
      (when
        (send-on-channel out prime nil)
        (go (filter-multiples in transfer prime))
        (sieve transfer out)))))
```

The function prime-generator returns a channel from which prime numbers can be received.

```
(setq prime-generator
  (lambda ()
    (let
      ((out (make-channel 0)))
      (progn
        (go (sieve (integer-generator 2) out))
        out))))
```

8. Discussion

8.1. Implicit Parallelism

The first example in chapter 7.1. (p-hanoi) implements a divide and conquer algorithm in an elegant, clean and functional way. When the function is run on a CPU with four hardware threads (Intel Core i5 M540), it reaches a speed up above factor 3 compared to its sequential counterpart (s-hanoi). Experiments with similar functions were run on CPUs with up to 8 hardware threads and showed a nearly linear speed up.

Regarding Amdahl's law [14], the higher the number of hardware threads, the smaller the sequential part of the program and its runtime has to be to reach a nearly linear speed up.

For a speed up of factor 64 the non-parallelizable part of a program must be smaller than 1.6%. This is in the range of time spent for garbage collection. Even if modern garbage collectors (like the Garbage-First collector [15]) work themselves with multiple threads they still interfere with the user program.

Experiments have to show, how well the implicit parallelism and the underlying Java virtual machine will scale on processors with that many hardware threads.

8.2. Channels favor an imperative programming style

Looking at the example for the use of channels (→ chapter 7.3.), the frequent occurrence of **when** and **progn** shows that the defined functions have a sequential programming style.

The reason for this is the heritage of the channel concept: Hoare's Communicating Sequential Processes [16]. Therefore the sequential nature of the example is not a surprise, because it uses communication primitives designed for sequential processes.

If the force to an imperative style when using channels is acceptable or a lack of elegance in a functional language is a matter of taste.

9. Outlook

The implicit parallelism in the Lisp Package can use the capabilities of state of the art consumer market processors with up to 8 threads. It will be interesting to see if the Lisp Package scales with upcoming hardware that supports more threads.

Unfortunately, it is not clear, if this hardware will actually ship. Intel's currently released consumer processors (based on Haswell microarchitecture) do not increase the thread count comparing with their predecessors [17]. There are predictions that the trend to more cores and hardware threads will stop [18].

For own experiments with the Lisp Package, a spreadsheet demo that uses the Lisp Package as engine to evaluate the formulas in cells can be downloaded from [1] (click the link "Calc.jar" at the end of the text). The JAR file also includes the source codes.

10. References

[1] http://www.simplysomethings.de/functional+programming/calc.html

[2] R. P. Gabriel, K. M. Pitman
"Technical Issues of Separation in Function Cells and Value Cells"

[3] Oracle Corporation
"Oracle Unveils SPARC T3 Processor and SPARC T3 Systems"
http://www.oracle.com/us/corporate/press/173536

[4] Intel Corporation
"Intel® Xeon Phi™ Coprocessor 5110P"
http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[5] J. Held, J. Bautista, S. Koehl et al.
"From a Few Cores to Many: A Tera-scale Computing Research Overview"

[6] M. McCool, A. D. Robinson, J. Reinders
"Structured Parallel Programming"

[7] B. Goetz et al.
"Java Concurrency in Practice"

[8] D. Chisnall
"The Go Programming Language Phrasebook"

[9] D. Flanagan, Y. Matsumoto
"The Ruby Programming Language"

[10] D. Pountain
"A Tutorial Introduction to Occam Programming"

[11] http://golang.org/ref/spec#Close

[12] http://en.wikipedia.org/wiki/Tower_of_Hanoi

[13] http://en.wikipedia.org/wiki/Dining_philosophers_problem

[14] G. Amdahl
"Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities"

[15] http://labs.oracle.com/jtech/pubs/04-g1-paper-ismm.pdf

[16] C. A. R. Hoare
"Communicating Sequential Processes"

[17] http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-desktops-brief.pdf

[18] H. Esmaeilzadehy, E. Blemz, R. St. Amantx, K. Sankaralingamz, D. Burger
"Dark Silicon and the End of Multicore Scaling"