PARALLEL AND SEQUENTIAL ALGORITHMS AND DATA STRUCTURES

LECTURE 2 SPARC and examples

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SYNOPSIS

- Functional Algorithms
- The Lambda Calculus
- The SPARC Language
  - Syntax and Semantics
  - Type System of SPARC
- Example
  - Threads, Concurrency, and Parallelism
  - Critical Sections and Mutual Exclusion

Introduction

- We define algorithms and data structures using nested parallelism in conjunction with a functional programming style
  - This is the best way to capture the core ideas of algorithms and parallelism in a concise, clear, safe, and precise way
  - will be useful <u>in a broad set of programming languages</u>

Nested parallelism

- Nested parallelism (or nested fork-join parallelism)
  - a style of parallelism in which any task can fork a set of new child tasks to run in parallel
  - When forking, the parent task suspends
  - when all the child tasks finish, they "join", and the parent continues
- sufficiently powerful to capture the parallelism in most of the algorithms needed for the purpose of this book
- For dynamic programming, we diverge slightly from this model to a somewhat more general model

Functional Algorithm

- a style of programming in which functions act like mathematical functions (a mapping from domain to a codomain, and no side effects), and can be uses as values (can be passed around, stored as data, and created on the fly)
  - have no side effects, parallelism is inherently safe and deterministic
  - allows powerful abstractions

SPARC

- We use a minimal, perhaps "toy", language called SPARC to describe algorithms and data structures.
  - SPARC has structures for supporting nested parallelism and supports only functional programming
  - is an extension of the lambda-calculus

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Function vs. Algorithm

- **Functions** are more accurately **algorithms**
  - mapping from inputs to outputs (**mathematical functions**)
  - specify the mechanism (code) by which <u>the output is generated from the input</u>

The Lambda Calculus

- developed *by Alonzo Church* in the early 30s, is arguably the first general purpose “programming language”
- Is a pure language, only supporting pure functions, and it fully supports higher-order functions
  - it is very simple with only three types of expressions, and one rule for “evaluation”
- It captures many of the core ideas of modern programming languages
- SPARC, is effectively an extended and typed lambda calculus

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Syntax and Semantics

- Definition (Syntax of the Lambda Calculus)
  - The lambda calculus consists of expressions *e*
    - a variable , such as x,y,z,…,
    - a lambda abstraction, written as (*λ x . e*), where *x* is a variable name and *e* is an expression, or
    - an application , written as (*e1 e2*), where *e1* and *e2* are expressions

华中科技大学
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Syntax and Semantics

- Definition (<u>Beta Reduction</u>)
  - For any application for which the left hand expression is a lambda abstraction, beta reduction “applies the function” by making the transformation:
  - $(\lambda\ x\ .\ e1)\ e2 \longrightarrow \quad e1[\ x/e2]$
  - <u>where e1[x/e2] roughly means for every (free) occurrence of x in e1, substitute it with e2</u>

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Church-Turing Hypothesis

- Story
  - In the early 30s, soon after he developed the language, Church argued that anything that can be "effectively computed" can be computed with the lambda calculus, and therefore it is a universal mechanism for computation
  - A few years later, Alan Turing developed the Turing machine and showed its equivalence to the lambda calculus that the concept of universality became widely accepted
- Church-Turing hypothesis: anything that can be computed can be computed with the lambda calculus, or equivalently the Turing machine
- Church-Turing complete: <u>any computational model that is computationally equivalent to the lambda calculus</u>

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Parallelism and Reduction Order

- the lambda calculus is inherently parallel
- Definition (Call-by-Value)
  - beta reduction is only applied to (λ x . e1) e2 if the expression e2 is a value, i.e., e2 is evaluated to a value (lambda abstraction) first, and then beta reduction is applied
- Definition (Call-by-Need)
  - beta reduction is applied to (λ x . e1) e2 even if e2 is not a value (it could be another application)
  - If during beta reduction e2 is copied into each variable x in the body, this reduction order is called call-by-name , and if e2 is shared, it is called call-by-need

Parallelism and Reduction Order

- **Call-by-value is an inherently parallel reduction order**
  - This is because in an expression (e1 e2) the two subexpressions can be evaluated (reduced) in parallel, and when both are fully reduced we can apply beta reduction to the results
- **call-by-need is inherently sequential**
  - In an expression (e1 e2) only the first subexpression can be evaluated and when completed we can apply beta reduction to the resulting lamba abstraction by substituting in the second expression
  - Therefore the second expression cannot be evaluated until the first is done without potentially changing which reductions are applied

SPARC

- SPARC is a parallel and "strict" functional language used throughout the book for specifying algorithms
- is formed from a mixture of several languages
  - ML class of languages : SML, Caml, F#
  - Mathematical notation
  - English descriptions

The SPARC Language: Syntax and Semantics

- **How to understand?**

```
listOfOne () =
let
   x = 1
   type t = Int
   type list = Nil | Cons of (t * list)
in
   Cons(x,Nil)
end
```

Syntax and Semantics

- Syntax
  - the structure of the program itself
  - A SPARC program is an expression
- Semantics
  - what the program computes
- operational semantics
  - how algorithms compute
- cost semantics
  - algorithms augmented with specific costs

Syntex Definition (SPARC expressions)

Identifier	id	$:= \ldots$	
Variables	x	$:= id$	
Type Constructors	$tycon$	$:= id$	
Data Constructors	$dcon$	$:= id$	
Patterns	p	$:= x$	variable
		$\mid (p)$	parenthesis
		$\mid p_1, p_2$	pair
		$\mid dcon\ (p)$	data pattern
Types	τ	$:= \mathbb{Z}$	integers
		$\mid \mathbb{B}$	booleans
		$\mid \tau\ [*\tau]^+$	products
		$\mid \tau \to \tau$	functions
		$\mid tycon$	type constructors
		$\mid dty$	data types
Data Types	dty	$:= dcon\ [\mathtt{of}\ \tau]$	
		$\mid dcon\ [\mathtt{of}\ \tau] \mid dty$	

Operations	op	$:= + \mid - \mid * \mid - \ldots$	
Bindings	b	$:= x(p) = e$	bind function
		$\mid p = e$	bind pattern
		$\mid \mathtt{type}\ tycon = \tau$	bind type
		$\mid \mathtt{type}\ tycon = dty$	bind datatype

Values	v	$:= 0 \mid 1 \mid \ldots$	integers
		$\mid -1 \mid -2 \mid \ldots$	integers
		$\mid \mathtt{true} \mid \mathtt{false}$	booleans
		$\mid \mathtt{not} \mid \ldots$	unary operations
		$\mid \mathtt{and} \mid \mathtt{plus} \mid \ldots$	binary operations
		$\mid v_1, v_2$	pairs
		$\mid (v)$	parenthesis
		$\mid dcon\ (v)$	constructed data
		$\mid \mathtt{lambda}\ p\ .\ e$	lambda functions
Expression	e	$:= x$	variables
		$\mid v$	values
		$\mid e_1\ \mathrm{op}\ e_2$	infix operations
		$\mid e_1, e_2$	sequential pair
		$\mid e_1 \mid\mid e_2$	parallel pair
		$\mid (e)$	parenthesis
		$\mid \mathtt{case}\ e_1\ [\mid p \Rightarrow e_2]^+$	case
		$\mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$	conditionals
		$\mid e_1\ e_2$	
		$\mid \mathtt{let}\ b^+\ \mathtt{in}\ e\ \mathtt{end}$	

Identifier

- In SPARC, variables, type constructors, and data constructors are given a name, or an **identifier**
  - An identifier consist of only alphabetic and numeric characters (a-z, A-Z, 0-9), the underscore character ("_"), and optionally end with some number of "primes"
    - E.g.: x', x1, xl, myVar, myType, myData, and my_data

Identifier	id	$:=$	$\ldots$
Variables	x	$:=$	id
Type Constructors	$tycon$	$:=$	id
Data Constructors	$dcon$	$:=$	id

Patterns

- variables and data constructors can be used to construct more complex patterns over data
  – For example, a pattern can consist of a pair (x, y) or a triple of variables (x, y, z), or it can consist of a data constructor Cons(x) or Cons(x,y)

$$
\begin{array}{llcll}
\textit{Patterns} & p & := & x & \textit{variable} \\
 & & \mid & (p) & \textit{parenthesization} \\
 & & \mid & p_1, p_2 & \textit{pair} \\
 & & \mid & dcon(p) & \textit{data pattern}
\end{array}
$$

华中科技大学 HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Built-in Types

- Types of SPARC include base types such as integers Z, booleans B, product types such as $\tau 1 * \tau 2 \ldots \tau n$, function types $\tau 1 \rightarrow \tau 2$ with domain $\tau 1$ and range $\tau 2$, as well as user defined data types

Types	τ	$:= \mathbb{Z}$	integers
		$\mid \mathbb{B}$	booleans
		$\mid \tau\ [*\tau]^{+}$	products
		$\mid \tau \rightarrow \tau$	functions
		$\mid \mathit{tycon}$	type constructors
		$\mid \mathit{dty}$	data types

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Data Types

- In addition to built-in types, a program can define ***new data types*** as a union of tagged types, also called variants, by "unioning" them via distinct data constructors
  - the following data type defines a point as a two-dimensional or a three-dimensional coordinate of integers

$$
\begin{array}{lcl}
\mathtt{type}\ point & = & PointTwo\ \mathtt{of}\ \mathbb{Z} * \mathbb{Z} \\
 & | & Point3D\ \mathtt{of}\ \mathbb{Z} * \mathbb{Z} * \mathbb{Z}
\end{array}
$$

Recursive Data Type

- In SPARC *recursive data types* are relatively easy to define and compute with
  - we can define a point list data type as follows

$$\texttt{type}\ plist = Nil \mid Cons\ \texttt{of}\ point * plist.$$

  - Based on this definition the list defines a list consisting of three points

$$Cons(PointTwo(0,0),\ Cons(PointTwo(0,1),\ Cons(PointTwo(0,2), Nil)))$$

Option Type

- ***Option types*** **for natural numbers can be defined as follows**

$$\texttt{type}\ option = None \mid Some\ \texttt{of}\ \mathbb{N}$$

- **option types for integers**

$$\texttt{type}\ intOption = INone \mid ISome\ \texttt{of}\ \mathbb{Z}$$

Values

- The irreducible units of computation include **natural numbers**, **integers**, **Boolean values**, **unary primitive operations**, **binary operations**, **constant-length tuples**, **data constructors** applied to values ……

Values	v	$::= 0 \mid 1 \mid \ldots$	integers
		$\mid -1 \mid -2 \mid \ldots$	integers
		$\mid$ true $\mid$ false	booleans
		$\mid$ not $\mid \ldots$	unary operations
		$\mid$ and $\mid$ plus $\mid \ldots$	binary operations
		$\mid v_1, v_2$	pairs
		$\mid (v)$	parenthesis
		$\mid$ $dcon\ (v)$	constructed data
		$\mid$ lambda $p\,.\ e$	lambda functions

Values

- As a functional language, SPARC <u>treats *all function as values*</u>
  - The anonymous function ***lambda p. e***
  - e is a function whose arguments are specified by the pattern p, and whose body is the expression e

- The function $\boxed{\text{lambda } x.\, x+1}$ takes a single variable as an argument and adds one to it.
- The function $\boxed{\text{lambda } (x, y).\ x}$ takes a pairs as an argument and returns the first component of the pair.

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Expressions

- Expressions, denoted by *e* and *variants* (with subscript, superscript, prime), are defined inductively
- infix expression : *e1 op e2*
  - Op include + (plus), − (minus), ∗ (multiply), / (divide), < (less), > (greater), or, and and
  - *f(e1,e2)*
- We use standard precedence rules on the operators to indicate their parsing
  - 3 + 4 * 5 = 3+(4∗5)
  - all operators are left associative unless stated otherwise
    - 5−4+2 evaluates to (5−4)+2=3 not 5−(4+2)=−1

華中科技大學

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Sequential and Parallel Composition

- two special infix operators: "," and "||"
  - comma operator or sequential composition
    - *(e1,e2)* lambda $(x, y).\ (x * x, y * y)$
    - evaluates *e1* and *e2* sequentially, one after the other, returns the ordered pair consisting of the two resulting values
  - parallel operator or parallel
    - *(e1 || e2)* lambda $(x, y).\ (x * x \,||\, y * y)$
    - evaluates *e1* and *e2* in parallel, at the same time, and returns the ordered pair consisting of the two resulting values
  - <u>The two operators are identical in terms of their return values</u>

Case Expressions and Conditinals

- A **case expression** such as

$$
\begin{aligned}
&\text{case } e_1 \\
&\mid \text{Nil} \Rightarrow e_2 \\
&\mid \text{Cons}\ (x, y) \Rightarrow e_3
\end{aligned}
$$

- **Conditionals**
  - *if-then-else expression*

Function Application

- A **function application** , *e1 e2*,
  - applies the function generated by evaluating e1 to the value generated by evaluating e2

$$(\text{lambda}\ (x, y).\ \ x/y)\ (8, 2)$$

$$(\text{lambda}\ (f, x).\ \ f(x, x))\ (plus, 3)$$

$$(\text{lambda}\ x.\ \ (\text{lambda}\ y.\ \ x + y))\ 3$$

= ?

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Bindings

- The *let expression*

$$\textbf{let } b^{+} \textbf{ in } e \textbf{ end},$$

  - *Varible binding*
  - *A function binding , x(p)=e*
  - Each *type binding* equates a type to a base type or a data type.

$$
\begin{array}{l}
\textbf{let} \\
\quad x = 2 + 3 \\
\quad f(w) = (w \times 4, w - 2) \\
\quad (y, z) = f(x - 1) \\
\textbf{in} \\
\quad x + y + z \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{let} \\
\quad f(i) \; = \; \textbf{if} \; (i < 2) \; \textbf{then} \; i \; \textbf{else} \; i \times f(i-1) \\
\textbf{in} \\
\quad f(5) \\
\textbf{end}
\end{array}
$$

Binding

let
$\quad$ type $point$ = $PointTwo$ of $\mathbb{Z} * \mathbb{Z}$
$\quad\quad\quad\quad$ | $PointThree$ of $\mathbb{Z} * \mathbb{Z} * \mathbb{Z}$
$\quad$ $injectThree\ (PointTwo\ (x, y)) = PointThree\ (x, y, 0)$
$\quad$ $projectTwo\ (PointThree\ (x, y, z)) = PointTwo\ (x, y)$
$\quad$ $compose\ f\ g = f\ g$
$\quad$ $p0 = PointTwo\ (0, 0)$
$\quad$ $q0 = injectThree\ p0$
$\quad$ $p1 = (compose\ projectTwo\ injectThree)\ p0$
in
$\quad$ $(p0, q0)$
end

type $tree$ = $Leaf$ of $\mathbb{Z}$ | $Node$ of $(tree, \mathbb{Z}, tree)$
$find\ (t, x) =$
$\quad$ case t
$\quad$ | $Leaf\ y \Rightarrow x = y$
$\quad$ | $Node\ (left, y, right) \Rightarrow$
$\quad\quad$ if $x = y$ then
$\quad\quad\quad$ return true
$\quad\quad$ else if $x < y$ then
$\quad\quad\quad$ $find\ (left, x)$
$\quad\quad$ else
$\quad\quad\quad$ $find\ (right, x)$

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Binding

- **Syntactic sugar for loops, recursion and while**

$$
\begin{array}{l}
\mathtt{sumSquares}(n) \ = \\
\quad \textbf{let} \\
\qquad (n, s) \ = \\
\qquad\quad \textbf{start} \ \ (n, 0) \ \ \textbf{and} \\
\qquad\quad \textbf{while} \ \ n > 0 \ \ \textbf{do} \\
\qquad\qquad s = s + n^2 \\
\qquad\qquad n = n - 1 \\
\quad \textbf{in} \ \ s \ \ \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{sumSquares}(n) \ = \\
\quad \textbf{let} \\
\qquad f(n, s) \ = \ \textbf{if} \ \ \mathtt{not} \ \ (n > 0) \ \ \textbf{then} \ \ (n, s) \\
\qquad\qquad\qquad\quad \textbf{else} \ \ \textbf{let} \\
\qquad\qquad\qquad\qquad\quad s = s + n \times n \\
\qquad\qquad\qquad\qquad\quad n = n - 1 \\
\qquad\qquad\qquad\qquad \textbf{in} \ \ f(n, s) \ \ \textbf{end} \\
\qquad (n, s) = f(n, 0) \\
\quad \textbf{in} \ \ s \ \ \textbf{end}
\end{array}
$$

Binding

Example 1.43. *The following is a recursive variant type defining binary trees.*

$$\textbf{type}\ \mathit{tree} = \mathit{Leaf} \mid \mathit{Node}\ \textbf{of}\ \mathit{tree} \times \mathit{int} \times \mathit{tree}$$

We can then create a tree using, for example

```
Node(Node(Leaf,1,Node(Leaf,2,Leaf)),3,Node(Leaf,4,Leaf))
```

How many notes?

```
countLeaves(T) =
  case T of
     Leaf => 1
   | Node of (L,R) => countLeaves(L) + countLeaves(R)
```

which corresponds to the tree:

```
          3
        /   \
      1       4
    /   \    /  \
  Leaf   2  Leaf Leaf
        / \
     Leaf  Leaf
```

counts the number of leaves of a tree recursively

SYNOPSIS

- Functional Algorithms
- The Lambda Calculus
- The SPARC Language
  - Syntax and Semantics
  - Type System of SPARC
- Example
  - Threads, Concurrency, and Parallelism
  - Critical Sections and Mutual Exclusion

Threads

- A *thread* , short for thread of execution： a computation that executes a given piece of code.
- *multithreaded* : A program that uses multiple threads
- two operations on threads
  - The operation ***spawn*** takes an expression, creates a thread to execute that expression, and returns the thread
    - Once spawned the thread starts executing concurrently with other threads in the program
  - The operation ***sync*** takes a thread and waits until that thread completes its execution

华中科技大学 HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Threads

$$
\begin{array}{l}
\texttt{let}\ \ t = \texttt{spawn}\,(\texttt{lambda}\ (\,)\,.\,\mathit{fib}\ n\,) \\
\qquad u = \texttt{spawn}\,(\texttt{lambda}\ (\,)\,.\,\mathit{fib}\ 2n\,) \\
\qquad ((\,),(\,)) = (\texttt{sync}\ t, \texttt{sync}\ u\,) \\
\texttt{in}\ \ (\,)\ \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{fib}\ x\ = \\
\quad \texttt{if}\ \ x \leq 1\ \ \texttt{then}\ x \\
\quad \texttt{else}\ \mathit{fib}\,(x-1) + \mathit{fib}\,(x-2)
\end{array}
$$

- Any question？ <u>has no way of communicating the result back</u>
  - <u>The *sync* operations ensure that the results are computed by waiting for the threads to complete</u>

$$
\begin{array}{l}
\texttt{let}\ \ (r, s) = (\texttt{ref}\ 0, \texttt{ref}\ 0) \\
\qquad t = \texttt{spawn}\,(\texttt{lambda}\ (\,)\,.\,r \leftarrow \mathit{fib}\ n\,) \\
\qquad u = \texttt{spawn}\,(\texttt{lambda}\ (\,)\,.\,s \leftarrow \mathit{fib}\ 2n\,) \\
\qquad ((\,),(\,)) = (\texttt{sync}\ t, \texttt{sync}\ u\,) \\
\ \texttt{in}\ \ (!r, !s) \\
\ \texttt{end}
\end{array}
$$

Thread Scheduler

- Multithreaded programs rely on a ***thread scheduler*** or scheduler for short, to execute the spawned threads to completion
  - At a given time, a scheduler can execute any subset of the spawned threads that are ready to execute, i.e., they are not waiting other threads

Concurrency and Parallelism

- *concurrency problem* if its specification involves multiple things happening at the same item
  - *Multithreading* is usually the technique of choice for solving concurrency
  - Using a *thread scheduler*, such multithreaded implementations can be made to work on sequential hardware, such as a computer with any number of processors

華中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Parallelism

- An algorithm is *parallel* if it performs multiple tasks at the same **time**
  - Both concurrent and non-concurrent problems typically accept parallel algorithms (solutions)

- *Concurrency* versus *Parallelism*
  - concurrency is a property of a "problem"
  - parallelism is a property of an implementation or a "solution"

Parallel Fibonacci

- The function *fib* below computes the *xth* Fibonacci number in parallel

```
fib x dest =
  if x ≤ 1 then
    dest ← x
  else
    let
      (da, db) = (ref 0, ref 0)
      a = spawn (lambda ( ) . fib (x − 1 ) da)
      b = spawn (lambda ( ) . fib (x − 2 ) db)
      (( ), ( )) = (sync a, sync b )
    in
      dest ← !da + !db
    end
```

```
fib x =
  if x ≤ 1 then
    x
  else
    let (ra, rb) = (fib (x − 1)) || (fib (x − 2)) in
      ra + rb
    end.
```

Mutable State and Race Conditions

- *Parallel* versus *Sequential Semantics*
  - We can think of any parallel SPARC program as a sequential program by replacing parallel tuples with sequential ones
  - *Sequential Elision*
    - obtained by replacing par (or ||) with simple sequential pairs
  - if the original parallel program is pure (purely functional), and does not use side effects, then corresponding sequential program is "*observationally equivalent" to the parallel one*

华中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Mutable State

- a multithreaded or parallel program uses *mutable state*
  - reasoning about its correctness and efficiency can become difficult

$$\begin{aligned}&\texttt{let } x = \texttt{ref } 0\\ &\quad ((),()) = (x \leftarrow 1) \parallel (x \leftarrow 2)\\ &\texttt{in print } x \texttt{ end}\end{aligned}$$

$$\begin{aligned}&\texttt{let } x = \texttt{ref } 0\\ &\quad ((),()) = (x \leftarrow x + 1) \parallel (x \leftarrow x + 1)\\ &\texttt{in print } x \texttt{ end.}\end{aligned}$$

Data race

- a data race occurs
  - multiple threads access the same piece of data and at least one of the threads update or write to the data
  - usually lead to *non-deterministic outcome*, which is in many cases an error condition
  - ***benign*** **:** <u>a data race does not impact adversely the correctness of a program</u>

Why Use Mutable State

- **Why use mutable state at all and why not program purely functionally?**
  - it is impossible to avoid mutable state completely
    - Even if a program is purely functional, it still has to allocate memory and write into it.
    - at some level of abstraction, we must operate on mutable state, even when it is “hiddden” behind the abstraction of purely functional programming
  - Mutable state enables implementing certain operations including purely functional ones more efficiently
    - updating a single position in an array requires copying an array if we are not allowed to mutate it

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Races are Considered Harmful

- <u>data races are harmful and should be avoided to the extent possible</u>
  - have 100 threads each of which execute 10 instructions that can cause races,
  - the total number of interactions we must consider are 10^{100} more than the number of atoms in the universe

Critical Sections and Mutual Exclusion

- a *critical section* is a part that cannot be executed by more than one thread at the same time
  - a critical section must be executed in mutual exclusion
  - typically contain code that alters data shared among parallel computations, and could lead to data races if executed concurrently

```
let
  debit bal delta =
    bal ← bal − delta
  credit bal delta =
    bal ← bal + delta
in
  (debit mybal 10) || (credit mybal 10)
end
```

Data Races and Critical Sections

- a *data race* could occur： a critical section is executed concurrently
  - When a data race occurs, the outcome of the program usually depends on the relative timing of events in the execution, and varies from one execution to another
  - It can be extremely difficult to find a data race
    - parallel programs usually exhibit ample non-determinacy in their execution, due to scheduling
  - Heisenbug
    - A data race may lead to an observably incorrect behavior only a tiny fraction of the time, making it extremely difficult to observe and reproduce it

Mutual Exclusion Problem

- **Solving： synchronizing between the threads by using synchronization instructions**
  - ***Spin locks*** **allow a thread to “busy wait” until the critical section is “clear” of other threads**
  - ***Blocking locks*** **allow a thread to “block” and wait for another thread to exit the critical section. When the critical section is clear, then the blocked thread receives a signal, allowing it to proceed. The term mutex , short for "mutual exclusion" is sometimes used to refer to a blocking lock**
  - ***Atomic read-modify-write instructions*** **, can read and modify the contents of a memory location atomically, allowing a thread to operate safely on shared data**

華中科技大学

HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Atomic Read-Modify-Write Instructions

- **Nonblocking Synchronization**
  - atomic read-modify-write instructions are called *nonblocking*
  - Nonblocking operations can be used to implement more complex concurrent *nonblocking data structures*
- ***Compare and Swap***
  - an atomic read-modify-write instruction that performs a memory read followed by a memory write atomically on a machine word
- ***Fetch and Add***
  - an atomic read-modify-write instruction that atomically updates the contents of a memory location and returns the contents (before the update)

Summary

- Functional Algorithms
- The Lambda Calculus
- The SPARC Language
  - Syntax and Semantics
  - Type System of SPARC
- Example
  - Threads, Concurrency, and Parallelism
  - Critical Sections and Mutual Exclusion