Chapter 3

SPARC: A Strict Language for Parallel Computing

To describe the algorithms covered in this book, we use a pseudocode notation that is based on a language, which we call SPARC. SPARC is a strict functional language similar to the ML class of languages such as Standard ML or SML, Caml, and F#. In pseudo code, we sometimes use mathematical notation, and even English descriptions in addition to SPARC syntax. This chapter describes the basic syntax and semantics of SPARC; we introduce additional syntax as needed in the rest of the book.

3.1 Functional Algorithms

Many of the algorithms in this book are *purely functional*. To understand what this means, let's review first what it means for an algorithm to be pure and functional. In our review below, we use an informal style.

We say that a computation has a ***side effect***, if in addition to returning a value, it also performs an effect such as writing to an existing memory location, printing on the screen, or writing to a file. For example, consider a function that given a natural number n as an argument computes and returns the n^{th} Fibonacci number, and also updates the argument n by overwriting it with the return value. This function has the side effect of changing the value of its argument. We say that a computation is ***pure*** if it doesn't perform any side effects. Pure computations return a value without performing any side effects. In contrast an ***impure*** or ***imperative*** computation can perform side effects. The Fibonacci function described above is impure. Pure computations correspond closely with mathematical systems or notation, where for example, determining the result of a calculation of formula does not affect the result of another. This notion of purity can be further extended to allow for effects that are not ***observable***. For example, the Fibonacci function described above may be implemented by using a mutable reference that holds some intermediate value that may be used to compute the result. If this reference is not visible to the outside world or is not used by any other computation, the function has no observable effect, and can thus be considered pure. Such effects are sometimes called

Remark 3.1. [The lambda calculus]
As with most functional programming languages, the ML class of languages are based on the ***lambda calculus*** (or λ calculus), a computational model developed by Alonzo Church in 1932. The lambda calculus is a very simple language consisting of expressions e which can only have three forms:

x,	:	a *variable name*,
$(\lambda\ x\ .\ e)$	:	a *function definition*, where x is the argument and e is the body, or
$e_1\ e_2$	:	a *function application*, where e_1 and e_2 are expressions;

and effectively only a single rule for processing expressions, called ***beta reduction***. For any function application for which the left hand expression is a function definition, beta reduction "applies the function" by making the transformation:

$$(\lambda\ x\ .\ e_1)\ e_2 \longrightarrow e_1[x/e_2]$$

where $e_1[x/e_2]$ roughly means for every (free) occurrence of x in e_1, substitute it with e_2. Computation in the lambda calculus consists of applying beta reduction until there is nothing left to reduce.

In the early 30s Church argued that anything that can be "effectively computed" can be computed with the lambda calculus, and therefore that it is a universal mechanism for computation. However, it was not until a few years later when Alan Turing developed the Turing machine and showed its equivalence to the lambda calculus that the concept of universality became widely accepted. The fact that the models were so different, but equivalent in what they can compute, was a powerful argument for the universality of the models. We now refer to the hypothesis that anything that can be computed can be computed with the lambda calculus, or equivalently the Turing machine, as the ***Church-Turing hypothesis***, and refer to any computational model that is computationally equivalent to the lambda calculus as ***Church-Turing complete***.

Although the lambda calculus allows beta reduction to be applied in any order, most functional programming languages use a specific order. The ML class of languages use call-by-value, which means that for a function application $(\lambda\ x\ .\ e_1)\ e_2$, the expression e_2 must be evaluated to a value before applying beta reduction. Other languages, such as Haskell, allow for the beta reduction before e_2 becomes a value. If during beta reduction e_2 is copied into each variable x in the body, this reduction order is called call-by-name, and if e_2 is shared, it is called call-by-need. Call-by-name is inefficient since it creates redundant computations, while call-by-need is both inherently sequential and not well suited for analyzing costs. In this book we, therefore, only use call-by-value. All these reduction orders are Church-Turing complete.

benign effects. This more general notion of purity is important because it allows for example using side effects in a "responsible" fashion to improve efficiency.

A programming language is called ***functional***, if it does not restrict the use of functions any more than the use of other values such as natural numbers. This principle is sometimes referred to as "functions as first-class values" or "functions as first-class citizens". Functional programming languages do not distinguish between a function and other values: both can be used as building blocks of values, stored in memory, and passed as arguments to functions, etc. For example, a function that returns the number `1` under some argument and the natural number `1` are the same. Treating functions as values leads to a powerful way to code. For example, we can write ***higher-order functions*** that take functions as their arguments.

Combining the concepts of purity and functional programming, the term ***purely functional algorithm*** refers to an algorithm that is both pure and functional, i.e., it is described in a functional language and it avoids use of observable side effects such as imperative updates to memory locations that can be observed and can thus have an effect on other computations. Purely functional algorithms are particularly useful for parallelism for two important reasons.

First, purely functional algorithms are safe for parallelism. In particular, parts of the algorithm may be executed in parallel without affecting each other. In contrast, in imperative programming, the programmer must take care that side effects don't alter the meaning of the program in unintended ways. The problem is that preventing side effects from altering the meaning of the program is very difficult, because 1) depending on the exact execution order (timing) of components, side effects may cause a computation to return different results at different times, and 2) there are exponentially many different orderings. For example, consider three tiny functions, 5-lines each, that read and write from the same memory location. To verify that these three functions execute correctly in parallel, the programmer may have to consider all of the $15!/(5!)^3 = 756756$, different orders in which the lines of these functions may interleave during evaluation. It is nearly impossible for any human being to comprehend such numbers of different possibilities. In a highly parallel system where there are usually many more than functions, this problem can quickly overwhelm any human programmer.

Second, higher-order functions (even in a language that is not pure) help with the design and implementation of parallel algorithm by encouraging the designer to think at a higher level of abstraction. For example, instead of thinking about a loop that iterates over the elements of an array to generate the sum, which is completely sequential, we can define a higher-order "reduce" function. In addition to taking the array as an argument, the reduce function takes a binary associative function as another argument. It then sums the array based on that binary associative function. The advantage is that the higher-order reduce allows for any binary associative function (e.g. maximum, minimum, multiplication, ...). By implementing the reduce function as a tree sum, which is highly parallel, we can thus perform a variety of computations in parallel rather than sequentially as a loop. In general, thinking in higher order functions encourages working at a higher level of abstraction, moving us away from the one-at-a-time (loop) way of thinking that is detrimental to code quality and to parallelism.

We note that coding a purely functional algorithm does not require a purely functional programming language. In fact, a purely functional algorithm can be coded in essentially any programming language—one just needs to be much more careful when coding imperatively in order to avoid errors caused by sharing of state and side effects. Some imperative parallel lan-

guages such as extension to the C language, in fact, encourage programming purely functional algorithms. The techniques that we describe thus are applicable to imperative programming languages as well.

Remark 3.2. [Race conditions]
Side effects that alter the result of the computation based on the evaluation order (timing) are called ***race conditions***. Race conditions make it difficult to reason about the correctness and the efficiency of parallel algorithms. They also make debugging difficult, because each time the code is run, it might give a different answer. For example, each time we evaluate a piece of code, we may obtain a different answer or we may obtain a correct answer 99.99% of the time but not always. There are several spectacular examples of correctness problems caused by race-conditions, including for example the Northeast blackout of 2003, which affected over 50 Million people in North America. Here are some quotes from the spokesmen of the companies involved in this event. The first quote below describes the problem, which is a race condition (multiple computations writing to the same piece of data). "There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the application processes, they were both able to get write access to a data structure at the same time [...] And that corruption led to the alarm event application getting into an infinite loop and spinning." The second quote describes the difficulty of finding the bug. "This fault was so deeply embedded, it took them [the team of engineers] weeks of poring through millions of lines of code and data to find it."

Remark 3.3. [Heisenbug]
The term ***Heisenbug*** was coined in the early 80s to refer to a type of bug that "disappears" when you try to pinpoint or study it and "appears" when you stop studying it. They are named after the famous Heisenberg uncertainty principle which roughly says that if you localize one property, you will lose information about another complementary property. Often the most difficult Heisenbugs to find have to do with race conditions in parallel or concurrent code. These are sometimes also called concurrency bugs.

3.2 The SPARC Language

We describe the syntax and the semantics of the core subset of the SPARC language. ***Syntax*** refers to the structure of the program itself, while ***semantics*** refers to what the program computes. Since we wish to analyze the cost of algorithms, we are interested in not just what algorithms compute, but how they compute. Semantics that capture how algorithms compute are called ***operational semantics***, and when augmented with specific costs, ***cost semantics***. Here we describe the syntax of SPARC and present an informal description of its operational semantics. We will talk about cost semantics in Chapter 4. While we focus primarily on the core subset of SPARC, we also describe some ***syntactic sugar*** that makes it easier to read or write code without adding any real power. Even though SPARC is a strongly typed language,

for our purposes in this book, we use types primarily as a means of describing and specifying the behavior of our algorithms. We therefore do not present careful account of SPARC's type system.

3.2.1 Syntax and Semantics

Definition 3.4 shows the syntax of SPARC. A SPARC program is an expression, whose syntax, describe the computations that can be expressed in SPARC. When evaluated an expression yield a value. Informally speaking, evaluation of an expression proceeds involves evaluating its sub-expressions to values and then combining these values to compute the value of the expression. SPARC is a strongly typed language, where every closed expression, which have no undefined (free) variables, evaluates to a value or runs forever.

Identifiers. In SPARC, variables, type constructors, and data constructors are given a name, or an ***identifier***. An identifer consist of only alphabetic and numeric characters (a-z, A-Z, 0-9), the underscore character ("_"), and optionally end with some number of "primes". Example identifiers include, x', x_1, x_l, $myVar$, $myType$, $myData$, and my_data.

Program ***variables***, ***type constructors***, and ***data constructors*** are all instances of identifiers. During evaluation of a SPARC expression, variables are bound to values, which may then be used in a computation later. In SPARC, variable are ***bound*** during function application, as part of matching the formal arguments to a function to those specified by the application, and also by **let** expressions. If, however, a variable appears in an expression but it is not bound by the expression, then it is ***free*** in the expression. We say that an expression is ***closed*** if it has no free variables.

Types constructors give names to types. For example, the type of binary trees may be given the type constructor `btree`. Since for the purposes of simplicity, we rely on mathematical rather than formal specifications, we usually name our types behind mathematical conventions. For example, we denote the type of natural numbers by $\mathbb{N}$, the type of integers by $\mathbb{Z}$, and the type of booleans by $\mathbb{B}$.

Data constructors serve the purpose of making complex data structures. By convention, we will capitalize data constructors, while starting variables always with lowercase letters.

Patterns. In SPARC, variables and data constructors can be used to construct more complex ***patterns*** over data. For example, a pattern can be a pair (x, y), or a triple of variables (x, y, z), or it can consist of a data constructor followed by a pattern, e.g., `Cons(x)` or `Cons(x, y)`. Patterns thus enable a convenient and concise way to pattern match over the data structures in SPARC.

Types. Types of SPARC include base types such as integers $\mathbb{Z}$, booleans $\mathbb{B}$, product types such as $\tau_1 * \tau_2 * \ldots \tau_n$, function types $\tau_1 \to \tau_2$ with domain τ_1 and range τ_2, as well as user defined 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***. For example,

Definition 3.4. [SPARC expressions]

Identifier	id	:=	$\ldots$	
Variables	x	:=	id	
Type Constructors	$tycon$	:=	id	
Data Constructors	$dcon$	:=	id	
Patterns	p	:=	x	variable
		|	(p)	parenthesization
		|	p_1, p_2	pair
		|	$dcon(p)$	data pattern
Types	τ	:=	$\mathbb{Z} \mid \mathbb{B}$	base type
		|	$\tau[*\tau]^+ \mid \tau \rightarrow \tau$	products and functions
		|	$tycon$	type constructors
		|	dty	data types
Data Types	dty	:=	$dcon$ [**of** τ]	
		|	$dcon$ [**of** τ] | dty	
Values	v	:=	$0 \mid 1 \mid -1 \mid 2 \mid -2 \ldots$	integers
		|	`true` | `false`	booleans
		|	`not` |...	unary operations
		|	$\wedge$ | `plus` |...	binary operations
		|	v_1, v_2	pairs
		|	(v)	parenthesization
		|	$dcon(v)$	constructed data
		|	$\lambda\, p\,.\, e$	anonymous functions
Expression	e	:=	x	variables
		|	v	values
		|	e_1 op e_2	infix operations
		|	e_1 , e_2	sequential pair
		|	$e_1 \,\|\|\, e_2$	parallel pair
		|	(e)	parenthesization
		|	**case** e_1 [| p **=>** e_2]$^+$	case
		|	**if** e_1 **then** e_2 **else** e_3	if then else
		|	$e_1\ e_2$	function application
		|	**let** b^+ **in** e **end**	local bindings
Bindings	b	:=	$x(p) = e$	function binding
		|	$p = e$	variable binding
		|	**type** $tycon = \tau$	type binding, synonyms
		|	**type** $tycon = dty$	type binding, datatype

the following data type defines a point as a two-dimensional or a three-dimensional coordinate of integers.

```
type point = Point2D of ℤ * ℤ
           | Point3D of ℤ * ℤ * ℤ
```

In SPARC recursive data types are relatively easy to define and compute with. For example, we can define a point list data type as follows

```
type plist = Nil | Cons of int * plist.
```

Based on this definition the list

```
Cons (Point2D (0,0),  Cons (Point2D (0,1), Cons (Point2D (0,2), Nil)))
```

defines a list consisting of three points.

Example 3.5. [Booleans] Some built-in types such as booleans, $\mathbb{B}$, are in fact syntactic sugar and can be defined by using union types as follows.

```
type myBool = myTrue | myFalse
```

Throughout the book, we use ***option*** types quite frequently. Option types for natural numbers can be defined as follows.

```
type option = None | Some of ℕ.
```

Similarly, we can define option types for integers.

```
type int_option = INone | ISome of ℤ.
```

Note that we used a different data constructor for naturals. This is often necessary for type inference and type checking. since, however, types are secondary for our purposes in this book, we are sometimes sloppy about ensuring proper type inference for the sake of simplicity. For example, we use throughout `None` and `Some` for option types regardless of the type of the contents.

Values and Expressions. Expressions describe the computations that can be expressed in SPARC. Evaluating an expression via the operational semantics of SPARC produce the value for that expression.

Values of SPARC, which are the irreducible units of computation include natural numbers, integers, Boolean values `true` and `false`, unary primitive operations, such as boolean negation `not`, arithmetic negation $-$, as well as binary operations such as logical and $\wedge$ and arithmetic operations such as $+$. Values also include constant-length tuples, which correspond to product types, whose components are values. Example tuples used commonly through the book include binary tuples or pairs, and ternary tuples or triples. Similarly, data constructors applied to values, which correspond to sum types, are also values.

As a functional language, SPARC treats all function as values. The anonymous function $\boldsymbol{\lambda}\, p\ \texttt{.}\, e$ is a function whose arguments are specified by the pattern p, and whose body is the expression e. For example, the function $\boldsymbol{\lambda}\, x\ \texttt{.}\, x\texttt{+1}$ takes a single variable as an argument and adds one to it. The function $\boldsymbol{\lambda}\, (x\texttt{,}\, y)\ \texttt{.}\, x$ takes a pairs as an argument and returns the first component of the pair.

Expressions, denoted by e and variants (with subscript, superscript, prime), are defined inductively, because in many cases, an expression contains other expressions.

An ***infix expression***, $e_1\ \texttt{op}\ e_2$, involve two expressions and an infix operator $\texttt{op}$. The infix operators include $+$ (plus), $-$ (minus), $\times$ (multiply), $/$ (divide), $<$ (less), $>$ (greater), $\vee$ (or), and $\wedge$ (and). For all these operators the infix expression $e_1\ \texttt{op}\ e_2$ is just syntactic sugar for $f(e_1, e_2)$ where f is the function corresponding to the operator $\texttt{op}$ (see parenthesized names that follow each operator above). We use standard precedence rules on the operators to indicate their parsing. For example in the expression

$$3 + 4 \times 5$$

the $\times$ has a higher precedence than $+$ and therefore the expression is equivalent to $3 + (4 \times 5)$. Furthermore all operators are left associative unless stated otherwise, i.e., that is to say that $a\ \texttt{op}_1\ b\ \texttt{op}_2\ c = (a\ \texttt{op}_1\ b)\ \texttt{op}_2\ c$ if $\texttt{op}_1$ and $\texttt{op}_2$ have the same precedence. For example

$$5 - 4 + 2$$

will evaluate to $(5 - 4) + 2 = 3$ not $5 - (4 + 2) = -1$ since $-$ and $+$ have the same precedence.

Expressions include two special infix operators: "," and "||", for generating ordered pairs, or tuples, either sequentially or in parallel. The ***comma*** operator "," as in the infix expression (e_1, e_2), evaluates e_1 and e_2 sequentially, one after the other, and returns the ordered pair consisting of the two resulting values. Parenthesis delimit tuples. The ***parallel*** operator "||", as in the infix expression $(e_1\ ||\ e_2)$, evaluates e_1 and e_2 in parallel, at the same time, and returns the ordered pair consisting of the two resulting values. The two operators are identical in terms of their return values. However, as we will see later, their cost semantics differ: one is sequential and the other parallel. The comma and parallel operators have the weakest, and equal, precedence.

Example 3.6. The expression

$$\boldsymbol{\lambda}\, (x\texttt{,}\, y)\ \texttt{.}\, (x\ \texttt{*}\ x\texttt{,}\ \ y\ \texttt{*}\ y)$$

is a function that take two arguments x and y and returns a pair consisting of the squares x and y. The expression

$$\boldsymbol{\lambda}\, (x\texttt{,}\, y)\ \texttt{.}\, (x\ \texttt{*}\ x\ \texttt{||}\ \ y\ \texttt{*}\ y)$$

is a function that take two arguments x and y and returns a pair consisting of the squares x and y by squaring each of x and y in parallel.

A ***case expression*** such as

case e_1
| `Nil =>` e_2
| `Cons (`x`,`y`) =>` e_3

first evaluates the expression e_1 to a value v_1, which must return data type. It then matches v_1 to one of the patterns, `Nil` or `Cons(x,y)` in our example, binds the variable if any in the pattern to the respective sub-values of v_1, and evaluates the "right hand side" of the matched pattern, i.e., the expression e_2 or e_3.

An ***if-then-else expression***, **if** e_1 **then** e_2 **else** e_3, evaluates the expression e_1, which must return a Boolean. If the value of e_1 is true then the result of the if-then-else expression is the result of evaluating e_2, otherwise it is the result of evaluating e_3. This allows for conditional evaluation of expressions.

A ***function application***, $e_1\ e_2$, applies the function generated by evaluating e_1 to the value generated by evaluating e_2. For example, lets say that e_1 evaluates to the function f and e_2 evaluates to the value v, then we apply f to v by first matching v to the argument of f, which is pattern, to determine the values of each variable in the pattern. We then substitute in the body of f the value of each variable for the variable. To ***substitute*** a value in place of a variable x in an expression e, we replace each instance of x with v. For example if function $\lambda\ (x, y)\,.\,e$ is applied to the pair $(2, 3)$ then x is given value 2 and y is given value 3. Any free occurrences of the variables x and y in the expression e will now be bound to the values 2 and 3 respectively. We can think of function application as substituting the argument (or its parts) into the free occurrences of the variables in its body e. The treatment of function application is why we callSPARC a ***strict*** language. In strict languages, the argument to the function is always evaluated to a value before applying the function. In contrast non-strict languages wait to see if the argument will be used before evaluating it to a value.

Example 3.7. The expression
$(\boldsymbol{\lambda}\,(x, y)\ .x/y)\ \ (8, 2)$
evaluates to 4 since 8 and 2 are bound to x and y, respectively, and then divided.

The expression
`(` $\boldsymbol{\lambda}\,(f, x)$ `.`$f(x, x)$ `)` $(\texttt{plus}, 3)$
evaluates to 6 since f is bound to the function `plus`, x is bound to 3, and then `plus` is applied to the pair $(3, 3)$.

The expression `(` $\boldsymbol{\lambda}$`x .(` $\boldsymbol{\lambda}$`y .x + y ) )` 3 evaluates to a function that adds 3 to any integer.

The ***let expression***, **let** b^+ **in** e **end**, consists of a sequence of bindings b^+, which define local variables and types, followed by an expression e, in which those bindings are visible. In the syntax for the bindings, the superscript $+$ means that b is repeated one or more times. Each binding b is either a variable binding, a function binding, or a type binding. Each ***variable binding***, $p = e$, consists of a ***pattern***, p, on the left and an expression, e, on the right. The expression is evaluated and its value is assigned to the variable(s) in the pattern. The value of the expression must therefore match the structure of the pattern. For example if the pattern

on the left is a pair of variables (x, y) then the expression on the right must evaluate to a pair. The two elements of the pair are assigned to the variables x and y, respectively. Each ***function binding***, $x(p) = e$, consists of a function name, x (technically a variable), the arguments for the function, p, which are themselves a pattern, and the body of the function, e. A let expression **let** b^+ **in** e **end** evaluates to the result of evaluating e giving the variable bindings defined in b. Each ***type binding*** can equate a type to a base type or a data type.

Example 3.8. In the following expression

let
  $x = 2 + 3$
  $f(w) = (w \times 4, w - 2)$
  $(y, z) = f(x - 1)$
in
  $x + y + z$
end

Line 2 binds the variable x to $2 + 3 = 5$; Line 3 defines a function $f(w)$ which returns a pair; Line 4 applies the function f to $x - 1 = 4$ returning the pair $(4 \times 4, 4 - 2) = (16, 2)$, which y and z are bound to, respectively (i.e., $y = 16$ and $z = 2$); and finally in Line 6 x, y and z are added giving $5 + 16 + 2$. The result of the expression is therefore 23.

We need to be careful about defining which variables each binding can see, as this is important in being able to define recursive functions. In SPARC the expression on the right of each binding in a **let** can see all the variables defined in previous variable bindings, and can see the function name variables of all binding (including itself) within the **let**. Therefore the function binding $x(p) = e$ is not equivalent to the variable binding $x = \lambda\, p\,.\, e$, since in the prior x can be used in e and in the later it cannot. Function bindings therefore allow for the definition of recursive functions. Indeed they allow for mutually recursive functions since the body of function bindings within the same **let** can reference each other.

Example 3.9. The expression:

let
  $f(i)$ = **if** $(i < 2)$ **then** i **else** $i \times f(i - 1)$
in
  $f(5)$
end

will evaluate to the factorial of 5, i.e., $5 \times 4 \times 3 \times 2 \times 1$, which is 120.

Syntax 3.10. [While loops] The ***while loop*** can appear as one of the bindings b in a **let** expression and has the syntax the following syntax.

$$
\begin{array}{l}
xs \;=\; \textbf{start}\;\; p\;\; \textbf{and} \\
\qquad\; \textbf{while}\;\; e_{continue}\;\; \textbf{do} \\
\qquad\quad b^{+}
\end{array}
$$

Here xs are the result variables holding the values computed by the while loop, the pattern p is the initial value for xs. Such a **while** loop evaluates by setting xs to pattern p and then evaluating the loop until $e_{continue}$ evaluates to `false`. In a typical use the body of the loop b^{+} defines the variables xs, whose final value will be the value of xs when the loop terminates.

We define the while loop syntax as equivalent to the following pair of bindings.

$$
\begin{array}{l}
xs \;=\; \textbf{start}\;\; p\;\; \textbf{and} \\
\qquad\; \textbf{while}\;\; e_{continue}\;\; \textbf{do} \\
\qquad\quad b^{+}
\end{array}
\quad\equiv\quad
\begin{array}{l}
f\;\; xs \;= \\
\qquad \textbf{if}\;\; \texttt{not}\;\; e_{continue}\;\; \textbf{then}\;\; xs \\
\qquad \textbf{else}\;\; \textbf{let}\;\; b^{+}\;\; \textbf{in}\;\; f\;\; xs\;\; \textbf{end} \\
\\
xs \;=\; f\;\; p
\end{array}
$$

Here xs, p, $e_{continue}$ and b^{+} are substituted verbatim. The loop is expressed as a function that takes xs as an argument and runs the body of the loop until the expression $e_{continue}$ becomes false at which time the variables xs is returned. The variables xs are passed from one iteration of the while to the next each of which might redefine them in the bindings. After the loop terminates, the variables take on the value they had at the end of the last iteration.

Loops. SPARC does not have explicit syntax for loops but loops can be implemented with recursion. Throughout the book, we use the following syntactic sugar for expressing while loops. Syntax 3.10 defines the syntax of the while loops. In the definition, the expressions $e_{continue}$ determines the termination condition for the loop, while the bindings b^{+} constitute the body of the loop. In a typical use, the body of the loop assigns to the variables of xs, effectively determining the return value of the while loop.

When evaluated a while loop starts by matching the variables xs to the pattern p and then continues to evaluate the while loop in the usual fashion. It first checks the value of $e_{continue}$, if it is false, then the evaluation completes. If not, then the bindings in b^{+}, which can use the variables xs, are evaluated. Having finished the body, evaluation jumps to the beginning of the **while** and evaluates the termination condition $e_{continue}$, and continues on executing the loop body and so on.

Example 3.11. The following code sums the squares of the integers from 1 to n.

```
sumSquares(n) =
  let
    (s,n) =
      start (0,n) and
      while n > 0 do
        s = s + n * n
        n = n - 1
  in s end
```

By definition it is equivalent to the following code.

```
sumSquares(n) =
  let
    f(s,n) = if not (n > 0) then (n,s)
             else let
                    s = s + n * n
                    n = n - 1
                  in f(s,n) end
    (s,n) = f(0,n)
  in s end
```

Example 3.12. The piece of code below illustrates an example use of data types and higher-order functions.

```
let
  type point = Point2D of ℤ * ℤ
             | Point3D of ℤ * ℤ * ℤ

  inject3D (Point2D (x,y)) = Point3D (x,y,0)

  project2D (Point3D (x,y,z)) = Point2D (x,y)

  compose f g = f g

  p0 = (0,0)
  q0 = project3D p0
  p1 = (compose project2D inject3D) p0
in
  (p0, q0)
end
```

The example code above defines a `point` as a two (consisting of x and y axes) or three dimensional (consisting of x, y, and z axes) point in space. The function `inject3D` takes a 2D point and transforms it to a 3D point by mapping it to a point on the $z = 0$ plane. The function `project2D` takes a 3D point and transforms it to a 2D point by dropping its z coordinate. The function `compose` takes two functions f and g and composes them. The function `compose` is a higher-order function, since id operates on functions.

The point $p0$ is the origin in 2D. The point $q0$ is then computed as the origin in 3D. The point $p1$ is computed by injecting $p0$ to 3D and then projecting it back to 2D by dropping the z components, which yields again $p0$. In the end we thus have $p0 = p1 = (0, 0)$.

Example 3.13. The following SPARC code, which defines a binary tree whose leaves and internal nodes holds keys of integer type. The function `find` performs a lookup in a given binary-search tree t, by recursively comparing the key x to the keys along a path in the tree.

```
type tree = Leaf of ℤ | Node of (tree, ℤ, tree)

find (t, x) =
  case x
  | Leaf y => x = y
  | Node (left, y, right) =>
      if x = y then
        return true
      else if x < y then
        find (left, x)
      else
        find (right, x)
```

Remark 3.14.
The definition

$$(\lambda\, x\,.\,(\lambda\, y\,.\, f(x,y)))$$

takes a function f of a pair of arguments and converts it into a function that takes one of the arguments and returns a function which takes the second argument. This technique can be generalized to functions with multiple arguments and is often referred to as ***currying***, named after Haskell Curry (1900-1982), who developed the idea. It has nothing to do with the popular dish from Southern Asia, although that might be an easy way to remember the term.

3.2.2 Type System of SPARC

Type me if you can.