PLang: A Concurrent Programming Language Using Shared Nothing-Threads

Jacob Hughes
Candidate 119386

Department of Informatics
Project Supervisor: Dr. Martin Berger
University of Sussex

April 2016

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text.

Signature:

Jacob Hughes

Acknowledgements

I would like to thank my project supervisor, Dr Martin Berger for his support throughout the development of this project, encouraging me when faced with relentless bug fixes, asking insightful questions about my design choices and offering invaluable advice.

Summary

As modern computer processors are reaching the limit of speed, programs which run concurrently are becoming more and more important. However, concurrent programs bring about many challenges which make programming much more difficult.

This dissertation introduces a brand new, concurrent programming language - PLang. In addition to this, a working compiler for PLang is provided supporting the language features described. The main areas covered in this dissertation are:

- An introduction to concurrency, programming languages, and computer architecture.
- Why PLang was created.
- Background information on compilation of concurrent programming languages.
- Requirements analysis for the implementation.
- How the PLang compiler was designed.
- PLang's memory layout at runtime
- An evaluation of PLang

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Aims

This project introduces PLang - A new programming language which supports concurrent programming. In addition, a fully functional compiler is introduced which can generate PLang programs. PLang differs from the more traditional mainstream implementations of concurrent languages in that it is based on the concept of shared-nothing threads. This is where threads are isolated from each other and do not share memory. Instead, these threads communicate by sending messages to each-other. (Explained in detail in Section 4.7.3). This project attempts to provide a solution to the problems associated with shared-memory concurrency described in the section below.

1.2 The Problem With Concurrency

In order to understand concurrency, one must be familiar with the basic architecture of a modern processor. Consider the high-level processor overview shown in figure 1.1. This multi-core processor can be understood as a collection of individual, isolated, mini-processors, known as cores. Each of these cores can execute instructions simultaneously. *L1* and *L2* caches inside the processor are small sections of memory. The level (L) of these caches refer to the speed in which they can be accessed by the core; this difference is primarily rooted in their physical proximity to the core. Memory in most processor architectures is accessed hierarchically. If data cannot be found inside either the L1 or L2 cache, it is accessed from the main memory, where the access time is several orders of magnitude slower.

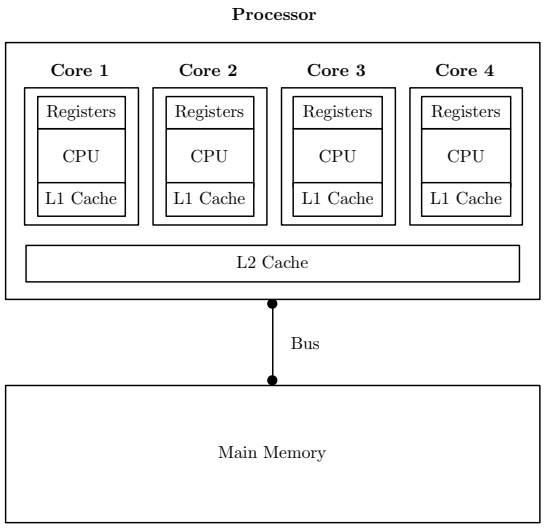


Figure 1.1: A modern multi-core processor.

Parallelism

When two or more sub units of a program are executed on different cores simultaneously, they are said to be running in parallel. Consider a simple real world example: A supermarket with more than one checkout is able to process many customers in parallel. (Figure 1.2).

Figure 1.2: Multiple checkouts in this store allow customers to be served in *parallel*.

Concurrency

Concurrency is a special case of parallelism and although both terms are commonly - and incorrectly - used interchangeably, their difference is subtle and based on their implementation of the division of labour. A concurrent program works by scheduling units of a program which can be executed out-of-order whilst its output remains determinable. This is somewhat analogous to somebody juggling objects (Figure 1.3).

This can be further extended to work on multiple cores, but its implementation allows for a concurrent program to run on a single thread, where the effects of calculations being executed simultaneously is simulated. The important distinction is that the sub-units of a concurrent program are not necessarily running in parallel.

Figure 1.3: This man is *concurrently* juggling balls.

Processes and Threads

Processes and threads are two basic units of computation. Although very similar, processes are considered to be isolated and have their own memory. Conversely, threads normally run within a process and memory is shared between many of them. In most implementations, threads are considered lightweight and faster than processes.

PLang implements very light-weight, isolated threads which share no memory. Although this approach encompasses elements of both OS defined threads and processes, a decision was made to name them processes to eliminate the possible implication that they share memory.

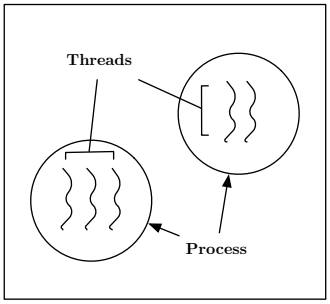


Figure 1.4: A diagram showing a program which contains both processes and threads.

As described in section 4.7, there are many approaches to achieving concurrency, each with their own advantages and disadvantages. Unlike sequential programs, where program flow is executed in the same logical order in which it is written, concurrency introduces significant additional design challenges. However, even with these difficulties, using concurrency in programming is highly beneficial to increasing throughput or masking program latency.

Challenges with Sharing Memory

Many of the challenges associated with concurrent programs can be eliminated if each thread is isolated, however, this allows for only trivial usage of concurrent programs. At some point during their execution, program threads may need access to memory from other threads. The most common way that this is implemented is using shared memory, which can be seen in programming languages such as Java and C++.

As explained in detail in Section 4.7, a shared memory approach risks having two or more threads attempt to write to a memory location simultaneously, which would leave it in an inconsistent state. This is known as a race condition. Consider the naively implemented stock control system in figure **??**. When a customer purchases milk at a checkout, the stock count for milk is decremented by 1 in the database. However, if two customers purchase milk simultaneously, a race condition occurs and one of the updated milk counts is overwritten making the updated milk total inconsistent.

In order to mitigate this, a programmer must carefully make use of locks

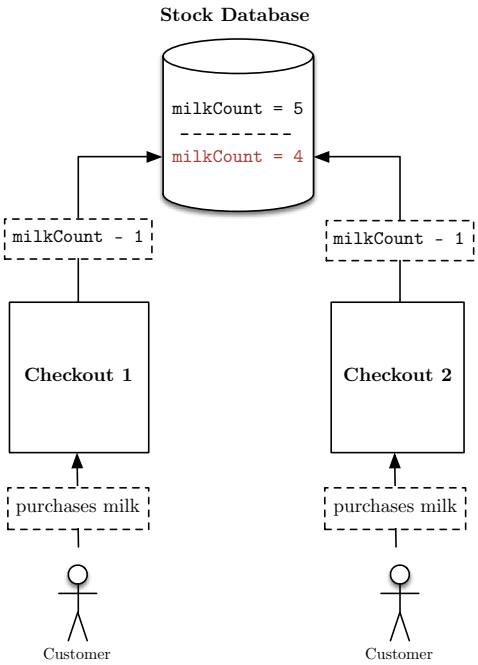


Figure 1.5: This man is *concurrently* juggling balls.

and thread synchronisation to ensure that only sequential access is provided to these critical sections of a program. This is notoriously difficult to implement correctly and can introduce hidden errors that are hard to identify and resolve. This can have devastating consequences, especially in safety critical systems that rely on correctly implemented concurrency. In 2003, a power outage in America affecting 50 million residents was caused by a race condition in the energy-management system which oversaw the distribution of power for 8 U.S states[11]. In addition, race conditions in concurrent languages can be exploited by malicious users in an attempt to compromise a program's security. In 2015, a researcher managed to exploit a transactional race condition in Starbucks' gift card system, effectively allowing them to steal money from the company.[13]. These issues cause programmers to program more defensively, which can incur not only an increase in development time, but also a performance overhead in their software. This report attempts to provide a solution to help mitigate this.

1.3 PLang - A Proposed Solution

This project introduces PLang, a statically typed (section 4.4.3), functional programming language (section 4.6) based on the concept of shared-nothing threads which communicate via message passing. PLang is heavily influenced by the Erlang programming language which uses a very similar approach. PLang differs, however, in that it requires no virtual machine and is a compiled language which outputs a program designed to run on the target architecture. This is considerably more difficult as it is working with a very low level of abstraction yet has the benefit of theoretical performance improvements. The PLang compiler designed in this project outputs programs designed to run on machines with the x86 architecture.

Given the development constraints of this project, PLang does not, and cannot not be expected to provide a fully optimised, comprehensive solution to shared-nothing concurrency. Instead, its main aim is to allow programmers to explore the advantages and disadvantages of the Actor Model while being protected from problems associated with traditional shared memory implementations, such as race conditions and deadlocks.

The name PLang is a portmanteau of Parallel-Language. Although concurrent-language would be more appropriate, it was rejected in order to avoid confusion with CLang, an existing C++ front-end compiler.

1.4 Target Users

PLang targets two key demographics: Researchers interested in modelling data on a simplistic actor model implementation and students who are interested in learning how to write concurrent programs which do not rely on shared memory. This is due, partly, to the current compiler's infancy. With considerable compiler optimisations and feature extension, it is possible that PLang would be a suitable candidate for lightweight multi-threaded applications in industry.

Chapter 2

Professional and Ethical Considerations

This project does not involve human participants as there is no interface that requires user testing. As such, there are no ethical considerations that need to be made and the project conforms to the BCS Code of Conduct.[5]

Chapter 3

Requirements Analysis

In this chapter, the requirements of PLang will be set out. The functional requirements specify what the software shall do whilst the non-functional requirements provide details on how it will be done.

3.1 Functional Requirements

In order to provide a language which is functional, concurrent, flexible and simple to use, the following implementation specific requirements must be met:

- Implement a lexer using the JFlex library
- Implement a parser using the CUP library, which allows for the generation of a LALR(1) parser to be created from the JFlex lexical tokens.
- Implement type inference based on the Hindley-Damas-Milner algorithm of type constraint generation and unification. [9]
- Implement higher order functions using the Scheme model of dynamic lexical scoping.
- Implement a stop and copy internal process garbage collector, which deallocates memory from no longer used first class function environments stored on the heap.
- Implement a reference counted garbage collector to deallocate memory usage of dead threads.
- Implement a user space thread manager in assembly which can schedule existing threads and adapt to the spawning and death of them during the program life cycle.

- Implement threads in the language which can be uniquely identified and referenced from the assembly driven scheduler. Threads should also be implemented with the reduction counter method of context switching seen in the Erlang model.
- Implement a command-line interface for the compiler that allows the user to supply the source code and target executable directory as command line arguments
- Threads must be able to communicate by sending messages to each other based on the Erlang style mailbox approach.

3.2 Non-Functional Requirements

- The language should be reliable and correct. (Evaluation of terms should be consistent)
- The generated program should be stable during runtime.
- The generated program should be portable across multiple Linux environments which support the *glibc* C library.

Chapter 4

Background Information

4.1 Memory Management

Before the process of compilation is explained, it is important to understand the layout of memory in a typical computer. This section explains the different types of program memory, and how they are individually managed.

4.1.1 Typical Memory Layout

Program memory is split into a stack and a heap. In most architectures, including Intel's x86, the stack conventionally grows downwards for historical reasons. The stack mainly is used to store the state of invoked function calls (commonly referred to as stack-frames or activation records). It is known at compile-time when these stack-frames must be cleared from memory and as a result the stack does not require garbage collection.

The heap is often implemented as a much more complex data structure than the stack and, because access is commonly not just accessed through the top value such as the stack, it has slower access times. The heap stores items which may outlive a stack-frame or whose scope may not be limited to a particular function. Common examples of this include: Complex data structures, global variables, higher order functions and objects.

It is not always known at compile-time when items should be removed from the heap. It is impossible to reason about the behaviour of program properties of a non-trivial program during runtime statically. This is rooted in Rice's theorem, which states that, for any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property.[12] And, as such, it cannot be determined by a compiler when this memory should be deallocated, thus memory

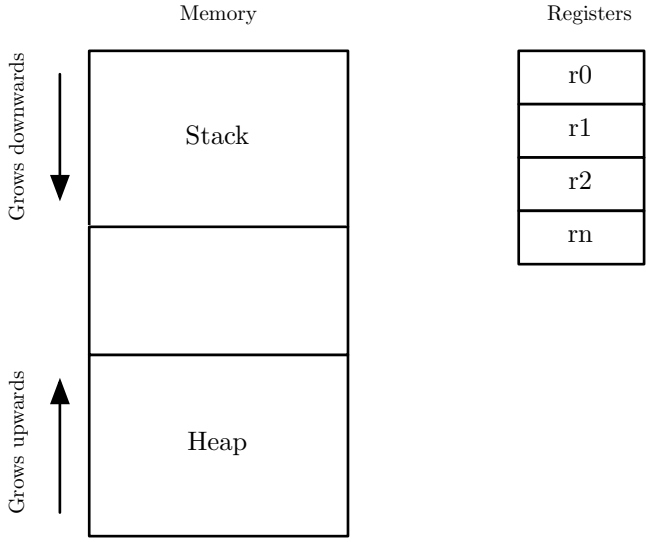


Figure 4.1: The typical layout of a program. Memory is split into a stack and heap.

must be managed manually by a programmer or garbage collected.

4.1.2 Manual Memory Management

In early implementations of programming languages, managing this heap memory was the responsibility of the programmer. This still remains in use today, with the C programming language being the most prevalent example. In this approach, a programmer would insert statements into their code when they wish to explicitly deallocate memory. This is advantageous to expert programmers who are able to make use of its potential performance benefits. However, this approach has a high chance of introducing memory leaks into a program which can make it error prone and difficult to debug.

Consider the following loop in the C programming language:

```
while(true) {
   str = (char *) malloc(1024);

}
```

This will loop until the program is forcibly terminated, allocating 1024 Bytes of memory at each iteration, and assigning a pointer to that memory to the variable named str. However, the scope of the str variable means that the pointer to each allocated memory block exists only until the next iteration. Without any form of deallocation, this program will eventually run out of heap memory. This is an extreme example of a memory leak.

4.1.3 Garbage Collection

A solution to this problem is garbage collection, where memory management is automated and controlled by the program during run time and abstracted away from the programmer. The purpose of a garbage collector to to reclaim memory so that it can be reused. Invented in 1959 for Lisp by John McCarthy [8], it has been hugely effective in reducing the number of programming errors resultant from memory leaks. Garbage collection does have some clear performance related disadvantages. The program will 'hang' until the heap has been reallocated. Not only does this slow it down, it can cause skips and pauses at non-determinate intervals. However, much research has been dedicated to efficient garbage collection and its performance overhead has been significantly reduced.

There have been many approaches to garbage collection developed over the years, this report will look at some of the most popular implementations.

Mark and Sweep

This was the first garbage collection algorithm created for reclaiming cyclic data structures from the heap. This algorithm can be thought of conceptually as running in two phases:

- The heap is scanned and a boolean mark bit is added for each object that is reachable (from a pointer in the program). In addition, fields for the object's size in memory and the location of each pointer are also added.
- The sweep phase traverses all available heap memory, freeing memory from items which are unmarked. In addition, items which are marked are reset to unmarked, ready for the next collection cycle.

Stop and Copy

Another common implementation of garbage collection is using the stop and copy algorithm. In this approach, the heap is split in half and data is written to just one half. Once this becomes full, the program stops its current execution and transfers only the live objects into the second heap fragment. The concept is very simple but implementation can be difficult as it must be known which items on the stack are pointers and which are immediate values. This process repeats for the rest of the program's life, each time a heap fragment becomes full. This can be shown in detail in figure 4.2. This approach is used in PLang to manage process heap memory.

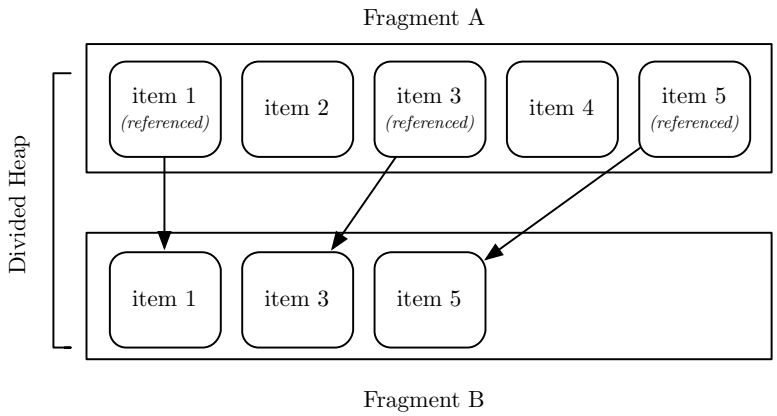


Figure 4.2: When fragment A of the heap is full, items which are still in use are copied to fragment B. No longer referenced items are discarded.

Reference Counting

Reference counting is a garbage collection technique that is faster and easier to implement than stop and copy. References are pointers to memory. The number of references to a particular item are counted and when this number reaches 0, the memory is unreachable and is therefore reclaimed. Reference counting is faster as the program does not need to halt to enable a garbage collection cycle to take place.

Unlike the stop and copy algorithm, this approach will not work with cyclic referencing. Consider the list in Figure 4.4a, where the head and the end of the list is pointing to the first list item. The first list item has a reference count of 2. When the head is removed and no longer references the first item in 4.4b, it still has 1 reference. The list items are no longer referenced externally and should be removed, it is referenced cyclically and thus can never be deallocated by the garbage collector.

Modern implementations of memory management work in parallel to actual program execution in order to prevent the noticeable latency when a garbage collection cycle takes place. This is very difficult to implement, as the safety of the heap state must be preserved in order to prevent race conditions upon access.

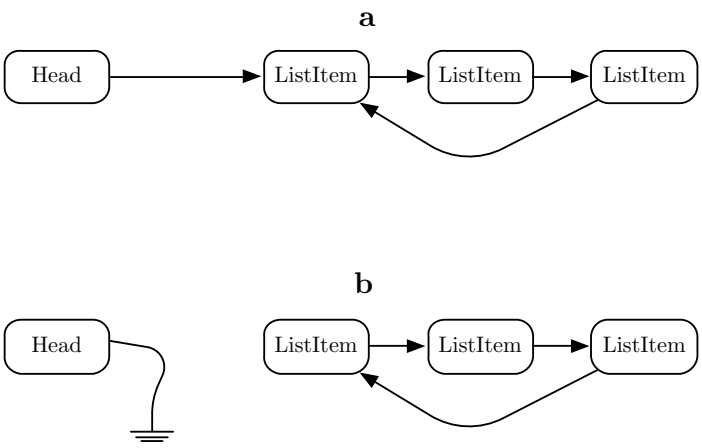


Figure 4.3: The problem incurred when trying to deallocate cyclic memory references.

4.2 Compiled and Interpreted Languages

Modern programming languages such as Java are highly abstracted. They allow programmers to write programs without worrying about low level concepts such as memory allocation or different chipset compatibility. In order for a programming language to run on a computer, the source code must be converted in some way which allows for it to be machine understandable. There are two main approaches to this, interpreting and compiling. Both of which have their own strengths and weaknesses which are orthogonal to the language specification and any input will always evaluate to the same result.

4.2.1 Interpreting the Source Language

An interpreter is a computer program that reads the input source code and executes the instructions one statement at a time. For each statement in the source code, an interpreter uses the following algorithm:

- **Parse:** The statement is read by the interpreter and checked for syntactic correctness. If the language is dynamically typed, the statement will also be type-checked here. Any errors found at this stage cause the interpreter to throw a runtime error.
- **Translate:** The statement is converted into executable machine instructions.
- **Execute:** The converted machine instructions of the statement are executed.

Interpreters are advantageous in that they are much simpler programs than compilers and are faster and easier to develop. However, the trade-off for this is that they are significantly slower than compiled languages.

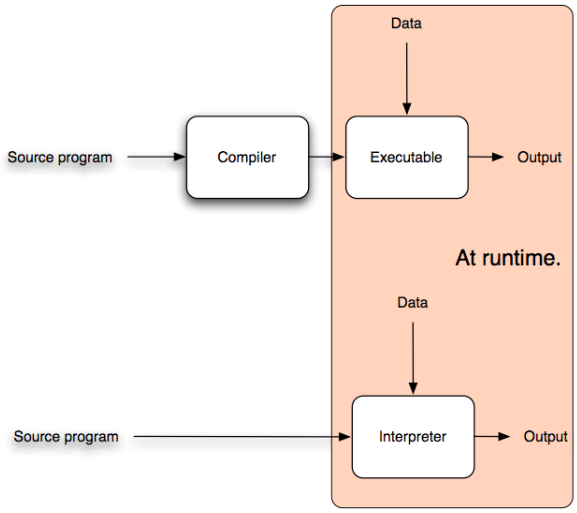


Figure 4.4: The difference between interpreted and compiled languages at runtime.[1]

4.2.2 Compiling the Source Language

A compiler is a computer program that reads the input source code and generates an executable program from it, which can be run later at any time. Although considerably more complex than an interpreter, it has several key advantages:

- It enjoys a much faster runtime speed at the cost of a one-off slow translation to the executable program.
- It can perform code optimisations at compile-time to further improve the runtime speed of the program.
- A program can be compiled once and the executable program can be ported to other machines with the same architecture.

PLang is a compiled language and in the interest of relevance, the topics and techniques mentioned in the rest of this report will be in the context of a language compilation.

4.3 The Pipeline of a Typical Sequential Compiler

A compiler takes an input, written in the source language and outputs a program which can be run on the target architecture. If the compiler discovers a problem with the source code, it will throw an error message and no output will be generated.

A standard compiler has a pipeline structure where each phase of compilation performs a different task on the given source language. The stages of the pipeline are described in 4.5. It is worth noting that each stage of compilation assumes the successful completion of the previous one.

Compilation is typically broken into two distinct parts: front-end compilation and back-end compilation. The front-end is concerned with converting the source input to a convenient representation for code generation, known as an Abstract Syntax Tree (AST). As explained below, the AST simplifies the process of code generation. Only when no errors are found in this stage, does the compiler back-end begin working on the AST to generate the output program.

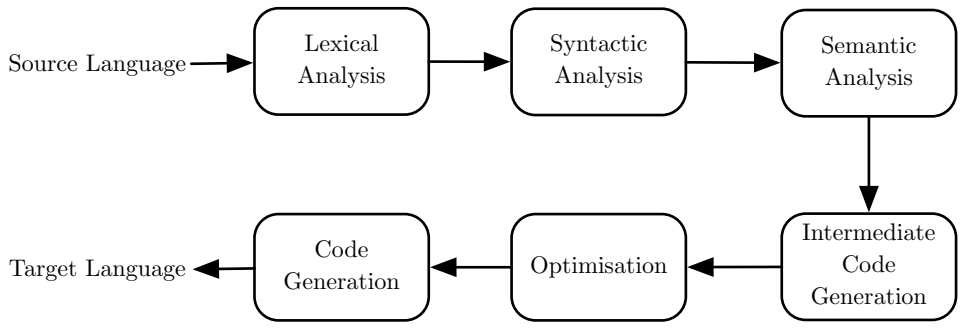


Figure 4.5: The pipeline structure of a compiler

4.4 The Compiler Front-End

4.4.1 Lexical Analysis

The first stage of compilation strips the source language of comments and whitespace. It then splits the remaining language into a series of lexical tokens. For example, the small program:

```
if(x==y)
        return true
else
        return false
```

Could be represented by the series of lexical tokens:

```
if, l_paren, identifier(x), equals, identifier(y), r_paren,
return, boolean_literal(true), else, return,
boolean_literal(false)
```

Most industrial strength lexers are built using lexer generators, where regular expressions are used to define the tokens in the language. These generators are implemented using table driven finite state automatas. Information regarding this is available in most compiler literature and is widely covered.

Compilation will halt if lexical errors are detected at this point. Lexical errors would consist of the inclusion of unrecognised symbols. Lexical analysis is not specific enough to check for errors such as misspelling of keywords (such as `retturn` instead of `return`). This would be seen as an identifier by the lexer. Errors such as these are found in later stages of compilation.

4.4.2 Syntactic Analysis

There are two tasks performed by syntactic analysis: to check the syntactic correctness of the language and to build the abstract syntax tree (AST). Just as the previous stage of compilation was only concerned with lexical correctness, it is important to note that this stage checks its input on syntax alone. Even though, for example, the program `x = 1 + true` is semantically incorrect, it is still syntactically well-formed and can be parsed successfully at this stage.

A language is parsed according to its context-free grammar (CFG). An example CFG for a simple language allowing for the representations of binary expressions on integers would be:

```
P := E | ϵ
E := E + E | E * E | E - E | E / E | ( E ) | D
D := 0 | 1 | 2 | ... | 9
N := D | D N
```

Here, ϵ denotes the empty string.

This example grammar could give rise to the following binary expressions:

```
5
10 + 1
1 * (5 - 25)
((((9))))
```

The AST is important as it provides a data structure for representing the program code for the rest of the stages in the compiler pipeline. An AST omits unnecessary constructs from the original source language (such as braces, semicolons and parenthesis) as the nested structure is implicit. This tree-like data structure allows the program to be easily traversable and annotations can be added to it for later use by other stages of compilation.

Let us take our original program:

```
if(x==y)
        return true
else
        return false
```

and generate its AST representation:

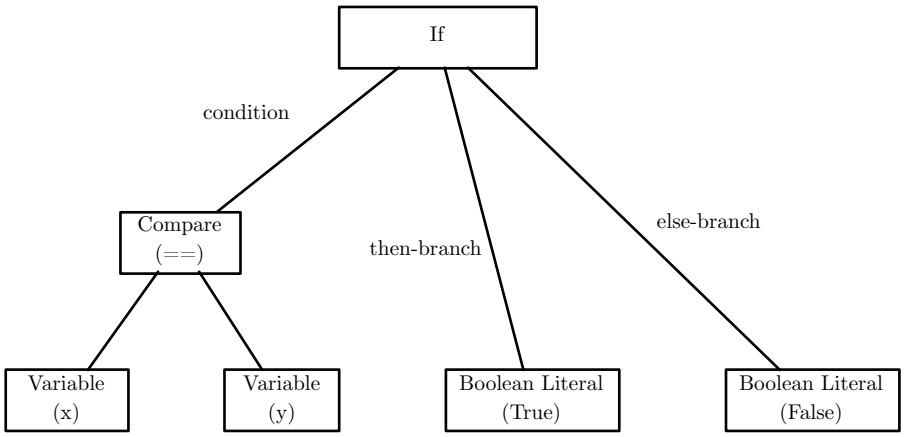


Figure 4.6: An example AST of a program containing a single if statement

4.4.3 Semantic Analysis

Most programming languages consist of some form of type system. This allows program constructs to be annotated with a data type. Types are a weak form of specification, and serve two main purposes: to solve simplistic programming errors[2] and are used for space allocation in code generation. The type of a construct denotes how it should behave and what operations can be performed on it.

Consider the following statement:

```
x = 5 + true
```

This statement does not make semantic sense, as it attempts to assign `x` to the result of the addition of an integer literal and a boolean literal. Typing allows us to catch such errors. Some languages exist which are *dynamically typed*. This means that type rules are checked at runtime. This is common in interpreted languages such as Python. However, a language is said to be *statically-typed*, if the typing system is enforced at compile-time.

The type constraints of a type system are written in the compiler. There are two main approaches to typing: type checking and type inference.

Type Checking

In this approach, a programmer must explicitly provide each variable in the program with a type declaration. The type checker then uses this information and type safety is verified by recursively traversing the AST to ensure that there are no contradictions in these constraints.

Consider the following code fragment with explicit type declarations:

```
int x = 10;
bool y = true;
int z = x + y;
```

This is an example of an ill-typed program. The type checker has a constraint in which the addition of boolean and integers is an illegal operation. When traversing the AST, the type checker rejects the third line, as `x` of type integer, cannot be added to `y` of type boolean.

Type Inference

This approach is more advanced than type-checking and was first implemented by Robin Milner in the ML programming language. As the name suggests, it seeks to infer the types of variables based on their usage. This has the advantage that the programmer does not have to provide explicit type declarations, thus reducing the verbosity of code. Perhaps the most

common example of type inference in a modern programming language is the Haskell typing system.

Type inference can be performed inductively as for each syntactic construct there is a very clear constraint as to what its type may be based on its components. Consider the small program fragment below, which consists only of a conditional:

```
if(x==y)
        return x + 1
else
        return y + 1
```

Due to the understanding that the conditional in an if statement must be a boolean type, we can infer that `x` and `y` must be the same time as they are checked for equality.

In the first `if` branch, `x + 1` is returned if the conditional holds true, this must mean that `x` is a numeric type because this type of binary addition can only be performed on numeric types. The same is true for `y` in the `else` branch.

However, consider a similar fragment where, instead, the `else` branch returns a boolean type:

```
if(x==y)
        return x + 1
else
        return false
```

A type inference algorithm would throw an error when evaluating this if statement and it would be considered ill-typed. This is because, one of the constraints is that the `then` branch and `else` branch of an if statement must be of the same type.

A working type inference algorithm known as the Hindley-Damas-Milner algorithm or algorithm W, was developed in 1982.[4] Milner's type inference was originally developed to provide a typing system for the lambda-calculus - A formal system for expressing the computation of functional application and abstraction. The lambda-calculus is a universal model of computation which is equivalent to a Turing machine.[16] Functional programming languages such as Haskell are rooted in the lambda calculus, and as such, the type

system can be expanded with additional constraints so that these languages can also be inferred. PLang is a type inferred language which is based off of this Hindley-Damas-Milner algorithm.

Critics of type inference claim that the lack explicit type declarations reduce the self documentation of code and make it harder to understand. As a result of this, some programming languages allow for optional type declarations which help the programmer convey the semantic meaning of certain functions or expressions.

4.5 The Compiler Back-End

4.5.1 Intermediate Code Generation

It is possible, at this stage, for the compiler to generate a program written in the target architecture (such as x86). This is the case for simple compilers such as PLang. However, this is not ideal for industrial strength compilers. The assembly language of many processor manufacturers may change regularly with each new update and rewriting the compiler backend to account for this involves a great deal of manpower and is expensive. Instead, an idealised representation of the target architecture, such as LLVM[7], is used, which can later be translated to the target architecture. This has the additional advantage of allowing the intermediate translation to be ported to machine code for multiple different chip architectures.

The AST is used to dynamically generate assembly code. This process is recursive and, for each symbol in the AST, a set block of assembly instructions is generated. The memory layout of most computer architecture consists of a stack and a heap. This can be seen in figure 4.1. In addition to this, a computer also contains registers.

When a function call is made, the code generator creates a stack-frame (or activation record) which is pushed on top of the stack. This houses the function's actual parameter values and a return address. The return address allows the program to jump back to the caller function after the callee has been evaluated. Figure 4.7 shows an example stack-frame. By convention, arguments are pushed in right to left order. The base pointer points to the

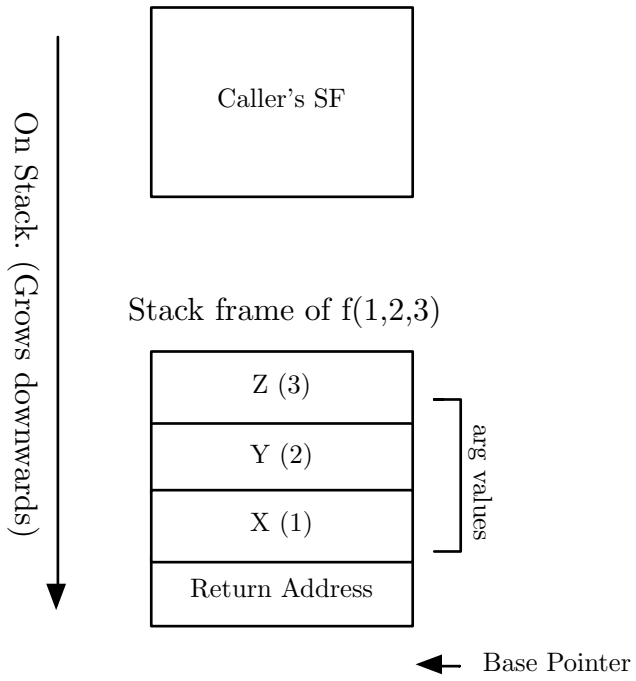


Figure 4.7: A stack frame for the function invocation f(1,2,3).

first item directly above the stack-frame. The base pointer (also known as frame pointer) is a register which is used to access a stack-frame's argument values at an offset to it. This is because it can be hard to reason about the position of the stack pointer at a particular point during runtime.

4.5.2 Optimisation

Modern industrial compilers such as the GNU C compiler contain hundreds of thousands of lines of code. The majority of which are for this optimisation stage of compilation. This is a large field of research which has seen vast improvements in the efficiency at which programs can be run sequentially on a CPU.

Before translating the intermediate code into the assembly language of the target architecture, an industrial compiler performs a variety of code optimisations. Programmers are generally advised against prematurely optimising their code for the sake of readability and the efficiency of development time. In addition, some inefficiencies in the code may have been introduced from the previous stages of compilation. During this stage an industrial compiler performs many optimisation techniques in order to try and mitigate this.

Unfortunately, it is very difficult to reason about how a program will behave at runtime statically. This limits the scope of what optimisation can be done statically whilst still guaranteeing the same program behaviour.

As PLang does not implement any optimisation techniques, compiler optimisation is only briefly covered in this report. However, one example of static optimisation is *loop unrolling*. In assembly languages, iteration is achieved by the use of jump instructions which move the instruction pointer to a different line in the program. This operation is very expensive for a CPU to perform and, as a result, loop unrolling calculates statically how many times a certain section of code is to be repeated and instead duplicates it. This is shown in the example below:

```
SET %reg1, 1
SET %reg2, 4
start_iter:           // begin iteration
       CMP %reg1, %reg2       // compare both registers
       JE fin_iter    // jump to finish if %reg1 = %reg2
       ADD %reg1, 1 // increment %reg 1
fin_iter:
       ....
```

When this loop is unrolled, it gives rise to the following assembly code:

```
SET %reg1, 1
SET %reg2, 4
ADD %reg1, 1
ADD %reg1, 1
ADD %reg1, 1
ADD %reg1, 1
```

In this simple example, the unrolled code is around the same size as before.

However, in most cases it would be much larger. This optimisation techniques has a trade-off between runtime efficiency and program size.

4.5.3 Final Code Generation

This stage is relatively straight forward, it converts the intermediate language into the target architecture after all the optimisations have been performed. This stage also links and loads any libraries that are used by the program so that the final program is executable. The semantic gap between the code from the intermediate language and the final target machine code is relatively small, hence the ease of this stage.

4.6 The Functional Paradigm

4.6.1 Higher Order Functions

An essential feature of functional programming languages is the support of *first-class functions*. A programming language is said to support this if it treats functions as *first-class citizens*. These terms were introduced by Christopher Strachey. He did not define the term strictly but referred to it when observing how the ALGOL programming language handles procedures:

> In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens - they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter). Stachey, [14]

A higher order function is a function which either takes a function as a parameter or returns a function. This is a powerful means of abstraction

which is rooted in the lambda calculus.

The most common example of this in practice is the implementation of the map function.

Consider the list:

```
li = [1,2,3,4,5]
```

Now, consider a trivial square function:

```
square(x) = x * x]
```

Calling `map square li` will return a new list, `[1,4,9,16,25]`. In this instance, the map function is a higher order function which accepts a function (`square`) as a parameter and applies it to each list item.

The example below shows a function which returns a function:

```
def makeBar() = {
        def bar(x,y,z) = {
                return x + y + z
        }
        return bar
}

def foo() = makeBar
foo(1,2,3) // returns 6
```

In this example, the function body of `makeBar` contains a function definition for the inner function `bar`. This inner function is then returned by `makeBar` and can be used within the rest of the program.

4.6.2 Difficulty of Implementation

Although being well understood for a long time, the problem with higher order functions is that they are difficult to compile. This is due to anonymous and nested functions implementing non-local variables. This is known as the funarg problem. It can be shown by the following example:

```
def foo(x,y) = {
        def bar() = {
                return x + y
        }
        return bar
}
```

In this code fragment, the parameters `x` and `y` are used in `bar` yet they belong to the parent function, `foo`.

This can cause problems as the function `bar` is returned by its parent. At this point, `foo` is no longer needed on the stack and its stack-frame is cleared, along with the variables `x` and `y`. When bar is then called, the stack or base pointer will not be able to locate the variable `x` or `y` as `foo`'s stack-frame no longer exists.

When higher order functions were introduced into Scheme, this problem of lexical scoping was solved by saving the arguments of the parent function in a stack-frame like structure on the heap, known as a closure. An 'up' pointer to this closure would then be included in each function's stack-frame. Function declarations can contain arbitrary nesting levels, and so each closure may contain a pointer its corresponding parent closure.

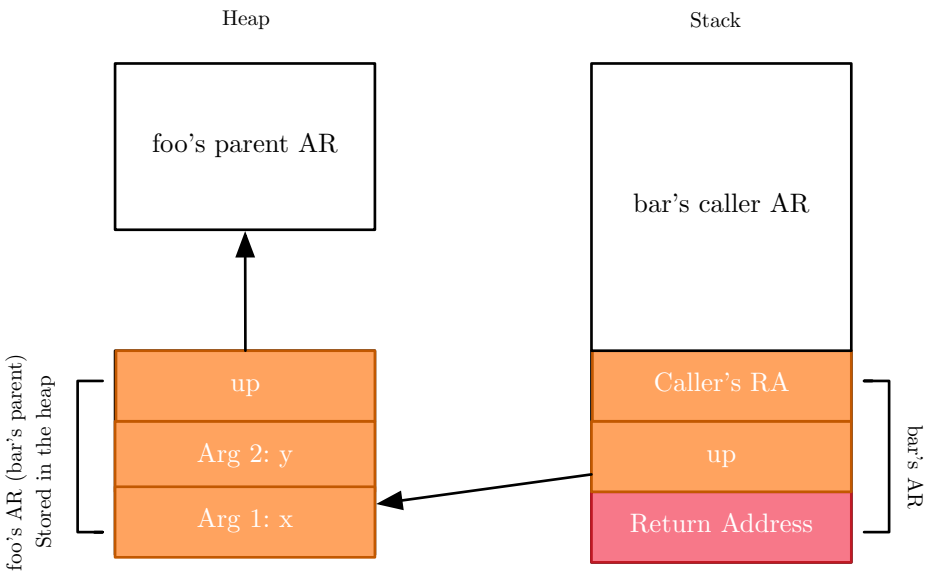


Figure 4.8: Dynamic lexical scoping using heap environments (AR = Activation Record)

4.7 Existing Approaches to Concurrency Implementation

4.7.1 User-space and Kernel-space Concurrency

The hardware in a computer is not directly accessible by a program. Multiple layers of abstraction exist in order to protect access to hardware and are provided by the operating system. This can be seen visually in figure **??**. The operating system kernel acts as the middle-man for computer hardware access. Applications in user-space which require hardware access must make a system call to the kernel, which then deals with the request.

Multi-threading can be achieved in two main distinct ways: user-space threading and kernel-space threading.

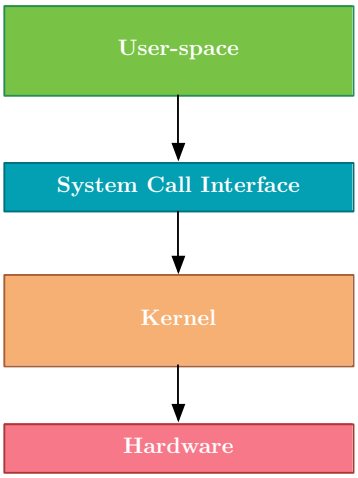


Figure 4.9: The layers of operating system abstraction

User-space Threading

When a concurrent application is run in user-space, no system calls to the kernel are required for a process to spawn threads. This allows for relatively inexpensive thread spawning and destruction. Kernel space thread spawning is expensive as operating system calls can be very time consuming. In addition, user-space multi-threading is much more portable as system calls are platform specific. This, however, means that the program must manage thread scheduling itself.

Kernel-space Threading

With kernel-space threading, the program threads are scheduled by the kernel and therefore make use of the operating system kernel's inbuilt scheduler. This allows for native support of parallelism on different CPUs, which is much more difficult to achieve in user-space. [15] The diagram below shows how these different methods of multi-threading are implemented on a computer:

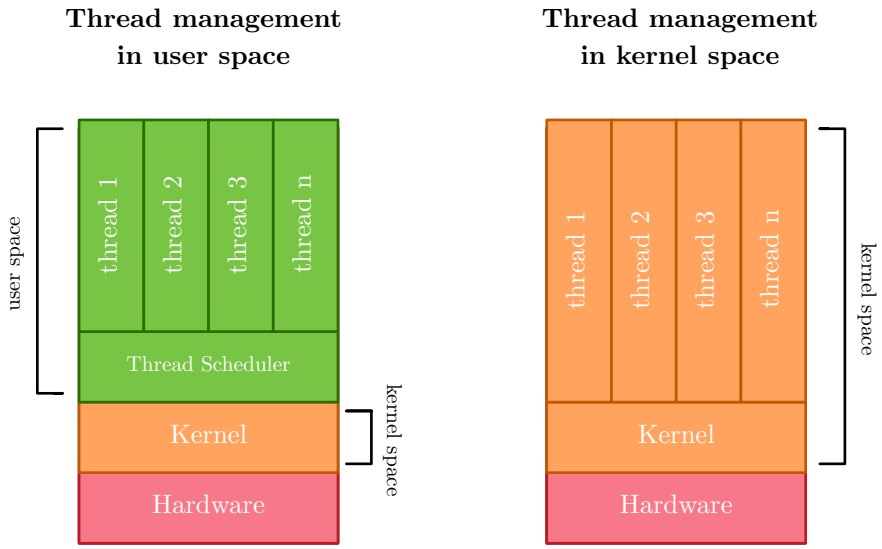


Figure 4.10: A comparison of thread management in user and kernel space

Thread Management

When threads are handled in user-space, threads are managed by a scheduler and given small time-slices to run on the processor which are rapidly switched between in order to create the illusion that they are running simultaneously - this is known as context-switching. This means that such a program can be successfully run on a uniprocessor system. There are many different ways one may choose to schedule threads, the most common being round robin or priority scheduling. Below shows an example of this scheduling achieved using the round robin approach.

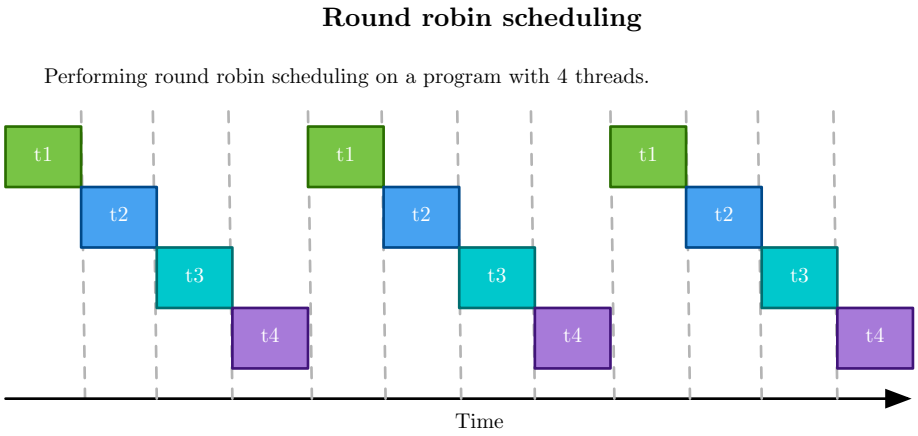


Figure 4.11: An example of round robin scheduling on 4 different threads

The Erlang Model of Reduction Counters

A key issue with this method of scheduling in user space is knowing when to make a context switch. If threads are switched based on a specific time increment, then a system call to the kernel must be made in order to determine the number of clock cycles passed. This is expensive and negates many of the efficiency benefits provided by user space abstraction.

The Erlang Programming Language resolves this by using the concept of thread specific reduction counters. Each thread is given a budget of 2000 reductions, and every time a thread performs a function invocation its reduction counter is decremented by 1. When a thread has exhausted all of its reductions, a context switch is performed. [3]. Erlang's only loop construct is recursion which means that its threads are guaranteed to context switch.

A reduction count of 2000 is relatively small and during a program life cycle will result in many context switches which makes it quite expensive to break up threads with high throughput. However, Erlang does this because of its favourability for low latency which is achievable when context switches are performed more frequently.

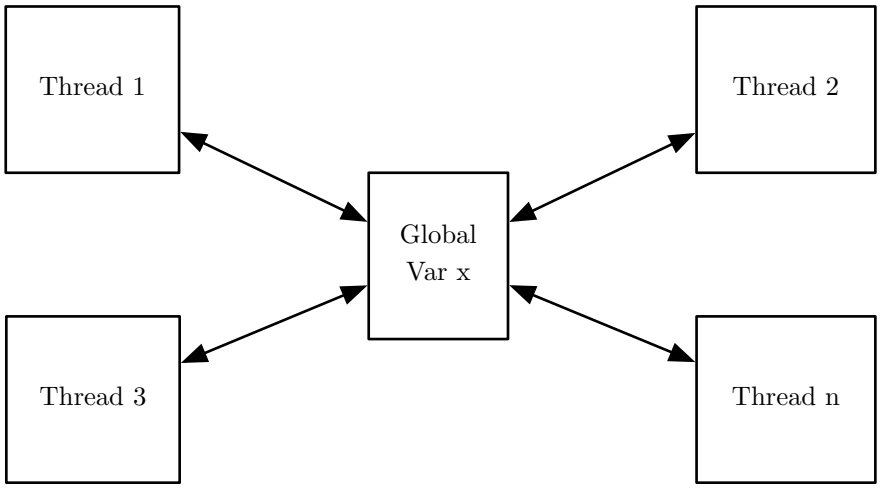


Figure 4.12: A diagram visualising multiple threads reading and writing to the same shared memory

4.7.2 Shared Memory

It is possible for threads to work autonomously, but most implementations allow for some form of communication. The most common approach to this is using shared memory. Conceptually, this approach is relatively simple - there exists some globally accessible memory which each thread has read and write access to. When a thread mutates the state of a variable in this memory, its new state is accessible to the other threads.

There exists a major drawback to this approach. If two or more threads attempt to write to the same space in memory simultaneously, only one of the writes will be performed and the others will be discarded. Figure 4.13 shows how this can happen. If theses synchronisation issues are not dealt with, shared memory can introduce race conditions which are extremely difficult to detect and replicate in debugging. Even if these issues are mitigated with defensive programming and memory locking, they can be an unwanted additional consideration when programming concurrently.

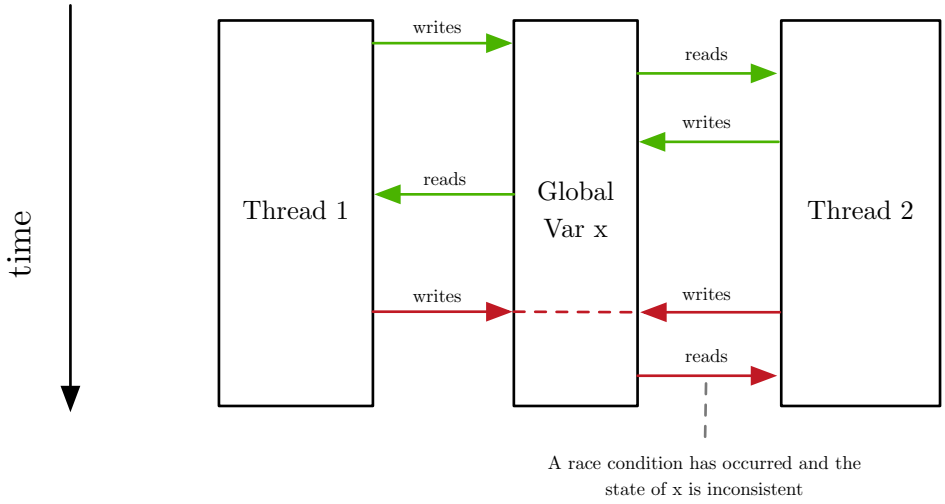


Figure 4.13: A diagram visualising the effect of a race condition when shared memory is not accessed atomically

4.7.3 The Actor Model

It is clear that the possibility of catastrophic consequences introduced with the shared memory approach make it a problematic way of producing thread safe programs. The Actor Model attempts to mitigate this by isolating threads having them share no memory. Instead, threads communicate by passing messages to each other. Perhaps the most famous example of this is the Erlang Programming Language.

Throughout much computer science literature, the term "process" is usually used when threads of execution share no data and the term "thread" when they do. Hence the shared-nothing threads of execution in languages such as Erlang and PLang are known as processes.

The Mailbox Design

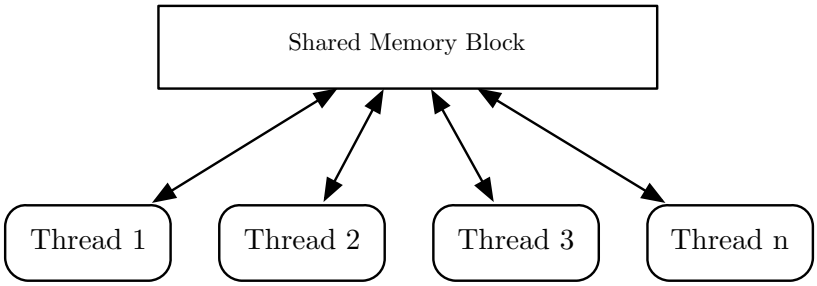
In these languages, each process contains a queue like data structure known as a mailbox. The mailbox is accessed with a first-in-first-out approach. When a process is sent a message, it appears in this mailbox awaiting future access. The nature of this design make race conditions impossible, guaranteeing thread safety. Consider figure 4.14 which shows a visual comparison of memory access in shared memory and actor based concurrency implementations.

Unfortunately, the mailbox approach introduces a new problem. The order of messages sent to a process is not guaranteed to be preserved. In a single threaded environment, messages sent from the same process can be guaranteed to arrive in order. The problem arises when messages are received from multiple processes, as shown in figure 4.15. One possible approach to mitigate this is to implement some form of pattern matching in the receiving . The pitfall of this is that runtime type checking would need to take place. The following pseudo-code gives an example as to how this could be implemented when matching on a message's source process ID (PID):

```
receive match pid{
        case 1 => doMethodPID1(payload)
        case 2 => doMethodPID2(payload)
        ...
}
```

This problem is exemplified in distributed systems that take place over remote networks. It is possible that messages transferred over the internet could be corrupted, arrive out of order, or not arrive at all. One must implement a message passing protocol, similar to TCP, to guarantee reliable message transfer.

a) Shared Memory



b) Shared-Nothing Message Passing

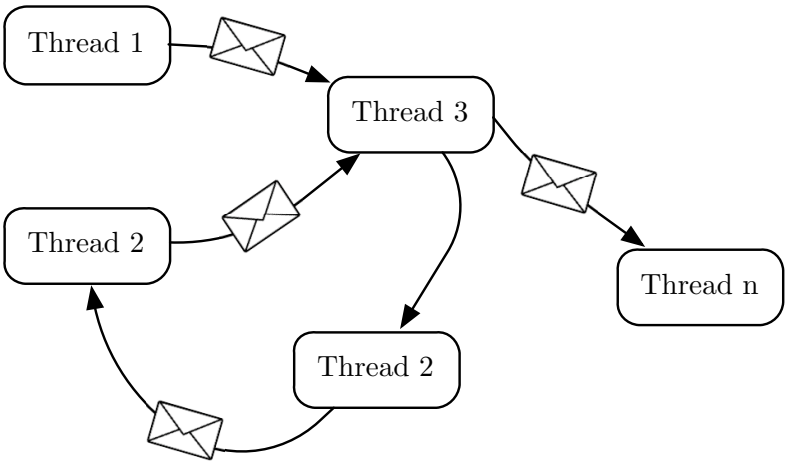


Figure 4.14: A comparison of shared memory and actor model implementations of concurrency.

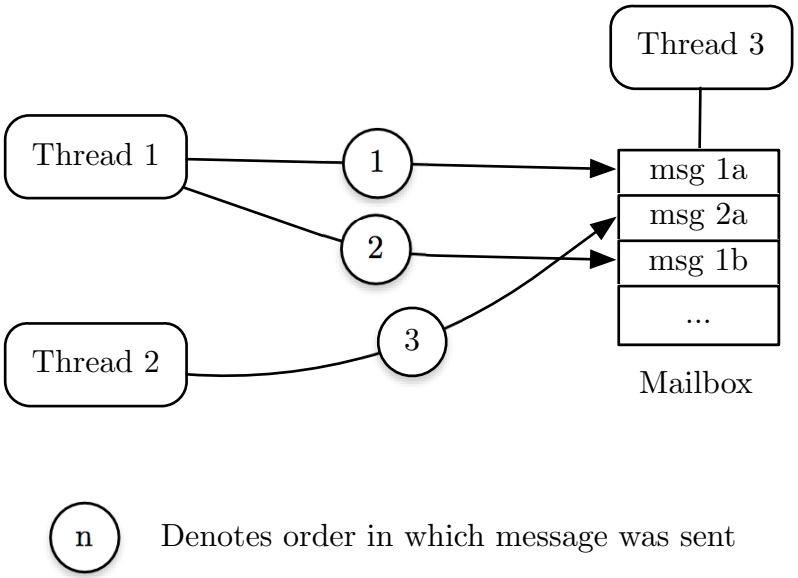


Figure 4.15: Receiving messages from multiple processes simultaneously.

Chapter 5

Using PLang

This chapter gives a brief overview of how to use PLang to construct runnable programs. The language is very simple to use and is heavily influenced from the succinct syntax of Scala and Python. For the full grammar of PLang please refer to the appendix.

5.1 Creating a PLang Program

5.1.1 PLang Files

PLang files end with the `.plang` file extension. The compiler does not accept any other file. All PLang programs must be written within one file and multi-file programs or packages are not yet supported.

5.1.2 A PLang Program

A valid PLang program is defined as one PLang file which contains all function declarations of the program followed by a nullary (zero argument) function named main as shown below:

```
def main() = {
 ...
}
```

5.2 Functions

Functions in PLang are declared with the `def` keyword, followed by the name of the function. Consider the declaration of the function, `myFunction` below:

```
def myFunction(x,y,z) = {
        ...
}
```

This function takes 3 formal parameters: x, y and z. Arguments in PLang are written in paired form and so must be comma separated and enclosed within parenthesis.

Below is an example of a PLang function declaration which takes no arguments:

```
def myNoArgFunction() = {
        ...
}
```

Note that in order to remain a valid function, the empty parenthesis are still required.

5.2.1 Everything as an Expression

Everything that can be written in PLang is an expression which can be evaluated to something. Unlike many imperative programming languages, there is no concept of statements. The programmer must be aware of this when using expressions, such as loop constructs, which typically are not known to return anything in order to avoid unintended behaviour.

There is no return statement inside PLang. Instead, the last expression inside a function to be evaluated is returned by the function, similar to the Scala programming language.

5.2.2 Function Invocation

Functions are invoked with their named followed by their actual parameters in paired format. PLang uses the Call-by-Value evaluation strategy. This is the most simple evaluation strategy to implement and also requires the least amount of reduction steps. This means that function arguments are evaluated to their normal form before use inside the function body. Consider the following examples:

```
myFunction(true,x,4)

f(1,(2+2+4)) // evaluates (2+2+4) to NF (8) before usage

myNoArgFunction()
```

Here, functions are invoked with their actual parameters. These must be written inside function declarations themselves, or in the main method. There is no global scoping in PLang.

5.2.3 Higher Order Functions

PLang supports the returning and passing of higher order functions. A higher order function is a function which takes another function as an argument, or returns another function as an argument.

Taking a Function as Argument

Consider the following example:

```
def add(x,y) = { x + y }

def apply(f,x,y) = {
        f(x,y)
}
```

In this example, the `apply` function takes a function parameter, `f`, and applies it to the other two arguments. This can be invoked with the following code:

```
apply(add,5,5) // returns 10
```

Here, the add function is passed to the apply function which performs an addition on the `x` and `y` variables.

Returning a Higher Order Function

When a higher order function is returned, it must be assigned to a variable in the calling function to be invoked separately. This must be done for each nested higher order function which is returned. Consider the following example:

```
def addArgs(x,y) = {
        def addAdditionalArgs(a,b) = {
                x + y + a + b
        };
        addAdditionalArgs
}

def addArgsUsage() = {
        returnedFunc = addArgs(1,2);
        returnedFunc(3,4) // invokes the func

}
```

In this example, `returnedFunc` is a variable which is assigned to a function returned from invoking `addArgs`. It can then be invoked later on within its local scope.

5.3 Processes

Threads of computation in PLang are referred to as processes. This was chosen instead of the term thread as processes are commonly understood not to share memory with each-other. Processes are the core of PLang. Everything you wish to run takes place from within one.

5.3.1 Spawning a Process

Processes are spawned using the `spawn` keyword. Process spawning takes one argument - the function invocation you wish to take place on the new process. For example, consider the function declaration below:

```
def f(x) = { print(x) }
```

which takes one argument, x, and prints it to the stdout of the program. One can invoke this on a separate process using the following expression:

```
spawn(f(1))
```

This creates a new process which, when run, will evaluate `f(1)`.

Process spawning returns the PID (process ID) of the new process. This can be assigned to a variable for use in passing messages to it later on. For example:

```
pidVal = spawn(f(1))
```

This assigns the PID, of type integer, to the variable named `pidVal`.

5.3.2 Sending Messages

Messages are sent in PLang by using the ! symbol, (known as the bang operator). The PID of the destination process should be placed to the left hand side of this operator, while the message should be placed on the right hand side. This gives the following structure:

```
pid ! message
```

A simple example of this in use would look like the following:

```
3 ! 6873 // Send process 3 the message 6873
```

A common usage of this would be to group it with the PID returned from spawning a new process:

```
pidVal = spawn(myFunc(1,2,3));
pidVal ! 5 // send the message 5
```

Note: If the process with the given PID cannot be found, a runtime error will occur, exiting the program. Error handling is usually better managed in more sophisticated industrial programming languages. However, error handling in concurrency is a complicated problem, and outside the scope of the PLang project.

5.3.3 Receiving Messages

Processes contain a mailbox which receive messages in first-in-first-out order. This can be thought of as a queue like data structure. When the `receive` keyword is used, it is evaluated to the value of the next message in the process mailbox. For example, consider the following function body is run on a process which has been sent the messages; 5 and 10:

```
x = receive; // the next msg in the mailbox is 5. x = 5
y = receive; // the next msg in the mailbox is 10. y = 10
x + y        // returns 15
```

If messages are received from the same process, their order will be guaranteed. However, if the process is receiving messages from multiple processes there is no guarantee that order will be maintained and this must be considered by the programmer.

Note: The receive action is blocking. If no messages are sent to this process while it is waiting it will block indefinitely.

5.4 Compiling and Running a PLang Program

The PLang compiler is a command line program which takes two arguments, 'source file' and 'destination folder', respectively. It can be run using the `plang` command. The PLang compiler supports the following command line flags:

-r Set the reduction count level (default 2000).

-m Size of the heap when program is run (Determines how many processes can be spawned).

-d Debug mode (prints the program stack to the console after each context switch).

Example usages of the compiler are shown below:

```
plang test.plang ~/Desktop/
```

This runs the PLang compiler with the `test.plang` source file and creates an executable of the same name on the user's desktop.

Chapter 6

The Compiler Implementation

The PLang compiler is written in Scala, there were several reasons for this decision:

- Scala supports both the objected-oriented and functional paradigm. Many elements of the compilation process are recursive and are naturally suited to a functional approach.
- Scala code runs on the Java Virtual Machine. As a result of this, all libraries one would expect to find in Java are available out of the box in Scala.

6.1 Lexing and Parsing

6.1.1 The Lexer

The first stage of PLang's compilation process is generated using JFlex. This is a lexer generator which generates Java code from its a given .flex file.

In the PLang compiler, the `scanner.flex` file is passed to JFLex which generates two outputs:

- `sym.java` A simple enum containing the lexical tokens to be used by the Parser
- `Lexer.java` The generated lexer. This uses a table driven automata, which is an implementation of the finite state machine described in Section 2.2.1)

In addition to producing an output in native Java code, JFlex was designed to integrate seemlessly with CUP, the Parser generator used by PLang in the next stage of compilation.

Table 6.1: **Note:** Order of precedence decreases with each row

Operator	**Meaning**	**Associativity**
=	Assignment	Right to left
+	Plus	Left to Right
-	Minus	
++	Concatenation	
==	Equals	Left to Right
<	Less than	
>	Greater than	
<=	L.T or Equals	
>=	G.T or Equals	
!	Bang	Left to Right

6.1.2 The Parser

Similar to JFlex, CUP is an open source program for Java which generates a parser from a specified grammar. The generated parser can construct an AST for a PLang program with syntactically correct input.

Precedence and Associativity of Operators

On its own, the PLang grammar is ambiguous. In order to resolve this, CUP requires an explicit declaration of operator precedence and associativity. PLang uses an order of precedence described in table 6.1.

The AST

PLang's AST is made up of Scala case classes, which were designed for extensive pattern matching. The full signature of these classes can be found in the appendix. These classes are used in the next stages of compilation to represent the program.

6.2 Type Inference

PLang is a strongly typed language which supports type inference based on the Hindly-Damas-Milner algorithm for the typed lambda calculus. The PLang type system was influenced by an existing F# implementation.[10]. At this point, PLang supports two base types: integers and booleans. The modular design of the typing system allows it to be easily extended for new constructs.

6.2.1 The PLang Typing Algorithm

The typing algorithm used by PLang works by applying constraints to each type construct and then unifies them. The following important points document how the whole program is typed:

- The AST representation of each function is traversed in the order in which they were declared.
- For each function, a placeholder type is created. If a function has multiple arguments, its type information is curried, making the type of PLang functions the result of chained unary function types.
- Assignments of variables are added to a type environment - a map of variable names to types.

The type inference algorithm can be best described by using the following pseudo-code. Note that the following is an extremely simplified example of the algorithm, and in the interest of conciseness, a small subset of the language expressions is included:

```
    def analyse(ast:AST, typeEnv:Map[String,Type]) = match ast {
            case Function(name,args,body) =>
                    returnType = analyse(body,typeEnv)
                    functionType = (curried(args) → returnType)
                    return functionType
            case Invoke(name,args) =>
                    invokeType = (curried(args) → metaType)
                    functionType = typeEnv lookup name
                    unify(invokeType, functionType)
                    return invokeType
            case Variable(name) =>
                    variableType = typeEnv lookup name
                    return variableType
            case Assign(name, rhs) =>
                    assignType = metaType
                    rhsType = analyse(rhs,typeEnv)
                    unify(assignType,rhsType)
                    return assignType
            case IntLiteral =>
                    return int
    }

    def unify(a:Type, b:Type):Type = {
            case (a: TypeVar, b: Constraint) => return b
            case (a: Constraint, b: TypeVar) => return a
    case (a: Constraint, b: Constraint) =>
            if (a == b) then return a else throw type error
}
```

In this algorithm, each symbol in the AST is traversed by the `analyse` function. This symbol is pattern matched (using case clauses) based on which type of symbol it is, e.g `Function, Invoke, Variable,` etc. A place-holder type is inserted, (α,β,γ,δ), where types are required.

When a constraint is found, such as an addition between two expressions, the types of those expressions are unified. This approach is carried out using

substitution. The most generic type constraint is substituted for the more specific type constraint. This can be seen in the example below:

```
x + 1
```

At this point, the specific type of variable `x` is not known, and has a place-holder type. However, `1`, is an integer literal and therefore is of type `int`. As this is the more specific type constraint, the place-holder type for `x` is substituted for `int` during unification.

Constraints cannot be unified if they are contradictory, for example:

$$\texttt{int} \neq \texttt{bool}$$

In instances such as this, the typing algorithm will throw a type error and the compilation process will be halted.

6.3 AST Conversion

The AST needs to undergo transformation in order to make the process of code-generation easier. There are several phases to this conversion stage. The purpose of each transformation will be explained during the relevant subsection.

6.3.1 Function Flattening

The first stage of AST conversion is to remove any nested functions and pull them out to the top level. This process is known as function flattening. The purpose of this is to simplify code generation for higher order functions. With all functions appearing as top level functions, much of the code for generation can be completed with slight alterations to function invocation generation.

Consider the following function declaration for `f(x,y,z)` in PLang, which returns a nested function `g`:

```
def f(x,y,z) = {
        def g(a,b) = {
                x + y + z + a + b
        };
        g
}
```

When its AST representation is converted by the compiler front-end, an AST representing the following is generated:

```
def f(x,y,z) = {
        return pair(f_g_decl_ptr, f_g_closure)
}

def f_g(a,b) = {
        return x + y + z + a + b
}
```

When pulled to the top level, the inner function is renamed with a prefix of its parent name. This is to prevent name clashes if there already exists a function named `g` at the top level.

In addition, a field for each variable's nesting level is added to its corresponding symbol in the AST. This is used for when functions are nested more than one level deep. It is used by the code generator to locate the lexical closure containing the variable's data.

Below shows an example of an inner function which is 3-levels deep:

```
def f(x,y) = {
        def g(a,b) = {
                def h(c,d) = {
                        x + y + a + b + c + d
                };
                h
        };
        g
}
```

When flattened, this will be translated to the following:

```
def f(x,y) = {
       return pair(f_g_decl_ptr, f_g_closure)
}

def f_g(a,b) = {
       return pair(f_g_h_decl_ptr, f_g_h_closure)
}

def f_g_h(c,d) = {
       return x + y + a + b + c + d
}
```

In this example, the nesting level of each variable is as follows:

```
x^2, y^2, a^1, b^1, c^0, d^0
```

Here, a nesting level of 0 denotes that the variable belongs to the function.

6.3.2 Assignment and Variable Translation

PLang allows programmers to define new variables inside functions. These new variables need to be stored in a place which is easily accessible for the code generator. As it can be determined statically how many new variables are assigned inside a function, they are appended to the formal function argument list and treated as arguments by the code generator.

As explained in the previous section, higher order functions require a pair of data, a pointer to the function declaration name and the closure pointer. This will require twice the amount of data on the stack-frame. Using the type information, variable symbols are converted to higher order function symbols.

During this traversal of the AST, all assignments are checked to see if they mutate existing variables or instantiate new ones. In the case of the latter, they are appended to their parent function's argument list ready for the next stage of AST translation. In addition, the variable translation is performed.

6.3.3 Offset Translation

Variable names exist in PLang as a convenient notation for programmers. They are not used within the code generator. Instead, an offset from the base pointer is used to access their value on the stack-frame. A variables offset is determined by its index position in the function declaration argument list.

In this stage of translation, the AST is traversed with all arguments and variables in the program are marked with an offset corresponding to their position in the argument list.

6.3.4 Invocation Padding

The symbol for invocation in PLang consists only of the function declaration name and a list of arguments. This argument list must be updated for the following reasons:

- In the previous stage, new variable assignments were appended to the function's formal argument list. Place-holders for these needed to be added in order for the stack-frame to be of the right size and order for the base pointer offsets.
- If the function being invoked is higher order, a place-holder for the parent's closure must be added to the stack-frame.

In this stage, a linear traversal of the AST is performed, where each symbol for invoke is checked for the above criteria and amended if necessary.

6.4 Code Generation (The Back-end)

The code generation phase of the compiler is modular and split into three entities with separate responsibilities:

Program Generation Responsible for setting up and initialising the program, generating common program functions, building the runtime scheduler and generating exit cases. The program generator is the main component of code generation.

Process Generation Responsible for spawning processes, saving process states, killing and reducing processes.

Function Generation Responsible for generating the code for function declarations which are ran inside processes. This is similar to the structure of a sequential compiler.

This process dynamically generates x86 assembly code for each symbol within the AST. The PLang code generator is thousands of lines of code long. It would be impractical to list the meaning of each assembly output for each symbol, instead, the next section provides an explanation of how the generated PLang program is structured and behaves at runtime.

Chapter 7

PLang's Runtime

This chapter describes the memory layout of a compiled PLang program at runtime. This section is rather technical and assumes deep background knowledge of compilation. If required, the rest of the report can be read fruitfully whilst ignoring this chapter.

7.1 The Program Layout

As explained in the background information section, the architecture of a generic computer architecture is composed of three main components; The stack, the heap and the registers.

Intel's x86 architecture contains a very limited number of available registers. These are as follows:

Table 7.1: Registers in x86

Name	**Purpose**
EAX	Accumulator register, holds return values.
EBX	General purpose.
ECX	General purpose, but commonly used as count register.
EDX	General purpose, but commonly referred to as data register.
EDI	General purpose.
ESI	General purpose.
ESP	Stack pointer.
EBP	Base pointer.
EIP	Instruction pointer. (Read only)

PLang's compiler can be thought of as an extended accumulator machine. This means that whenever possible, the compiled program will attempt to use only the accumulator register, `EAX`, for the sake of simplicity. Register use within the program is consistent throughout its runtime execution.

Figure 7.1 shows a high level overview of the program structure of a PLang program. The majority of PLang's complexity takes place in the stack, the heap is used only for storing processes which are not currently in execution. Every program is initialised with a main method in PLang, the first PLang process (PID 1) is set up as the process containing the code to be executed in the main method.

7.1.1 The Program Heap

The program heap is the most simple aspect of PLangs architecture. It contains only the *process store* which houses all processes in the program in their 'dirty state' while they await their turn for execution. The *process store* is of fixed size at runtime but its size can be specified by the programmer at compile time. The *process heap* size determines the maximum number of processes PLang is able to have in memory at once. Figure 7.2 gives a high level overview of the contents of the *process store*.

7.1.2 The Program Stack

This section describes in detail the different components of the main program stack. In order to avoid confusion, it is important to note that as the stack grows downwards, the top of the stack refers to the item which is next in line to be popped off. For this reason, diagrammatically, the top and bottom of the stack will appear to be inverted.

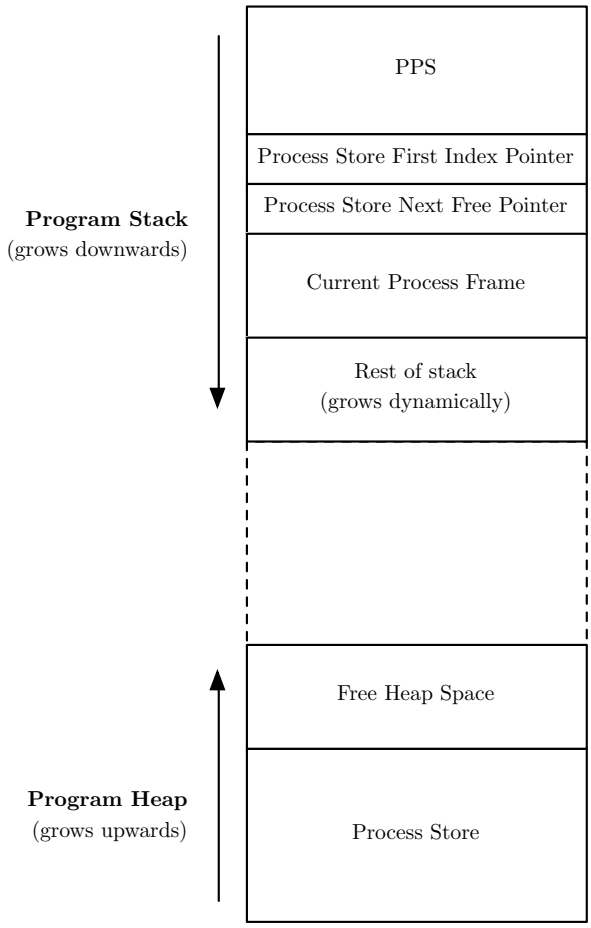


Figure 7.1: The layout of a PLang program in memory.

This section will describe each of the following items found on the program stack (figure 7.1), in detail:

- The PPS (Process Pointer Scheduler).
- The 'First index pointer' on the process store.
- The 'Next free pointer' on the process store.
- The *current process frame*.

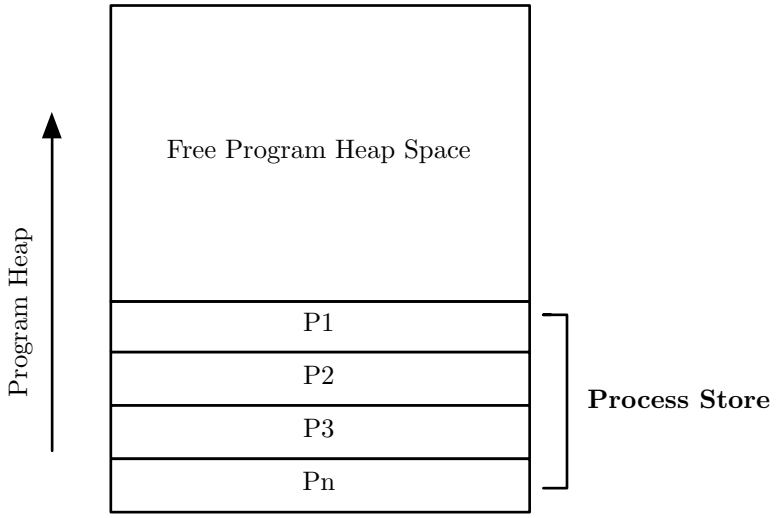


Figure 7.2: The process store, stored in the heap memory of the main program.

The Process Pointer Scheduler (PPS)

As shown in figure 7.1, a PLang program will always contain a Process Pointer Scheduler (PPS) at the bottom of the stack. The PPS is responsible for multiple aspects of a PLang program:

- As the name suggests, it schedules the various PLang processes in a round robin fashion.
- It contains a mapping of each process to its position in the process store.
- Its size determines the maximum number of processes that may exist at any point in time during the program lifecycle.
- It is responsible for exiting the program and handing control back to the OS if all processes have been executed.

Consider the diagram in figure 7.3. It shows a detailed overview of the contents of the PPS. The *next free pointer* will always point to an empty

block of memory available inside the PPS. This value is incremented every time a newly spawned process is saved on the process store. The next value in the PPS is the *current process pointer*, this points to the mapping inside the PPS for the process currently in memory. The rest of the PPS simply consists of pointers to each process on located in the process store. Any empty blocks inside the PPS are zero-valued.

In order to perform round robin scheduling, each time a context switch is performed (Explained later), the *current process pointer* is incremented to the next non-empty value inside the PPS. This may require looping to the top of the PPS if the bottom has been reached before locating the next process. If, however, no processes can be located and the *current process pointer* arrives back at its starting position, all processes have been executed and the program will exit.

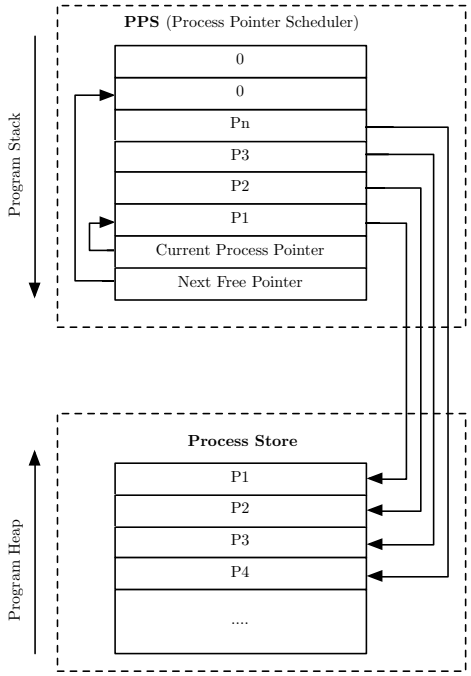


Figure 7.3: A detailed look at the Process Pointer Scheduler.

Current Process Frame

The current process in which is being run is stored here. The design of each process frame is explained in section 7.2.1. All process level execution is carried out when a process is in this position. This can be seen as analogous to a chopping board. Food is stored in the fridge (the process store), until it is placed on the chopping board (current process frame), where it is subsequently is chopped (executed).

Other Aspects of the Stack

Directly above the PPS are the first index and next free pointers for the process store. The *first index pointer* remains fixed during the duration of a PLang program and points to the address of the first position on the process store. The *next free pointer* is variable, and points to the next free space available on process store. This value is incremented each time a new process is stored on the process store.

Finally, after the *current process frame*, exists the rest of the stack which will grow and shrink dynamically. This is used for storing values in intermediate calculations by the program which are not related to the execution of any processes. This may include values used in context switching, garbage collecting and traversing memory.

7.1.3 Register Allocation

A stack is a Last in First Out (LIFO) data structure. Only the top element can be popped and in order to obtain direct access to other items stored inside (such as values inside the PPS or current process frame) they must somehow be referenced. To achieve this, one register is permanently set to a fixed address on the stack for the entire duration of the program lifecylce. This address points to the first position of the process callstack inside the current process frame. All processes in PLang are of the same fixed size, the PPS size is also fixed. For this reason the address of this offset register will always be correct. With this design, every single value below this register can be accessed with some offset to it. The chosen register was `EDI`.

7.2 The Process Layout

This section describes the core of PLang - its process architecture. Every action a programmer intends to compute is executed inside a process.

7.2.1 Process Design

Processes in PLang are designed to be very lightweight, the reasons for this are twofold:

- This permits the spawning of vast numbers of processes without quickly exhausting the available RAM.
- Context switching should be cheaper as less memory mapping is required.

Figure 7.4 shows an overview of a process in PLang, which consists of 4 main subsections:

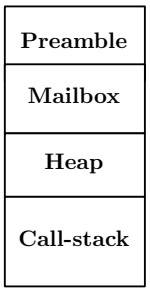


Figure 7.4: The layout of a process in PLang.

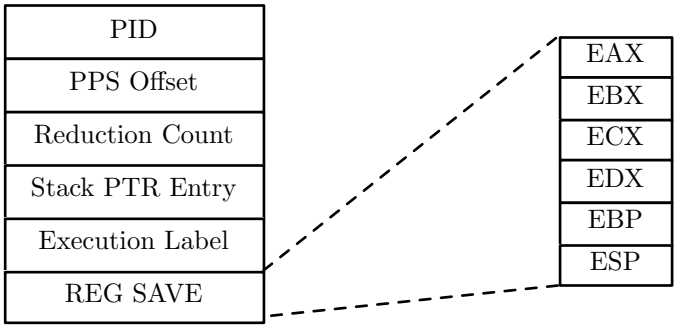


Figure 7.5: A detailed look at the contents of a process's preamble.

The Preamble

Figure 7.5 shows a high level overview of the preamble. Each field inside is exactly 4 Bytes in size.

PID The process ID, this is automatically generated when a new process is spawned and globally unique.

PPS Offset An address pertaining to the process's position inside the PPS

Reduction Count The number of steps left before a context switch (See 1.4)

Stack PTR Entry The value of the stack pointer before the process was loaded into the *current process frame*. This is important, as the process's own call stack manipulates the stack pointer and its old value must be maintained for when the process is finished.

Execution Label The address pertaining to the assembly code instruction in which the program must jump to when it begins to execute the process.

Reg. Save The state of the registers which were saved from when a context switch was last performed on the process.

The Heap

Processes have the ability to store data which may outlive its activation record on its call-stack. This small section of memory is called the heap (Not to be confused with the main program heap which contains the *process store*.) Figure 7.6 shows that this section of the PLang process divided into two parts in order to allow it to be garbage collected using the stop and copy algorithm.

The heap is access via the heap pointer, which increments to the next free value on the heap each time a new item is added. If the heap pointer gets to the boundary of the fragment it is in, program execution will halt whilst the internal process garbage collector is ran. In its current form, the only data stored on the heap are lexical closures for higher order functions.

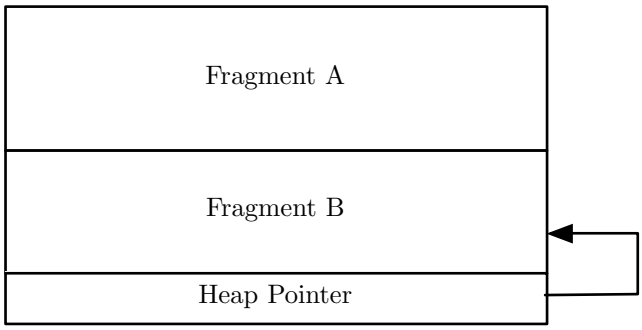


Figure 7.6: The heap memory inside a process.

7.2.2 The Life and Death of a Process

Spawning

Processes can be spawned from any other process within PLang. Whenever a new process is spawned, the following order of events take place immediately:

1. The state of the current process is saved.
2. The next available block in the PPS is given a pointer to the location of the new process's assigned space on the process store.
3. The new process is initialised with the stack frame of its passed function invocation placed on the process call-stack.
4. The new process is saved on the process heap.
5. The 'spawner' process is restored into the *current process frame*.
6. The PID of the newly spawned process is returned in the accumulator, allowing it to be assigned to a variable by the spawner.

This approach of switching the old and new processes in and out of memory is very similar to what happens during a general context switch. It is worth considering that due to the design of the PPS, appending the new process to the next available block (step 2) will not necessarily place the process at the end of the scheduler, as expected in most round robin scheduling.

Figure 7.7 shows the state of a newly initialised process.

Process Death

When the function invocation given to a process has been fully evaluated, the process is no longer needed and will die. A process's death is handled in the following order:

1. The process's mapping inside the PPS to its position inside the *process store* is zeroed.
2. A context switch occurs where the next process mapping in the PPS is placed in the *current process frame* ready to be executed.

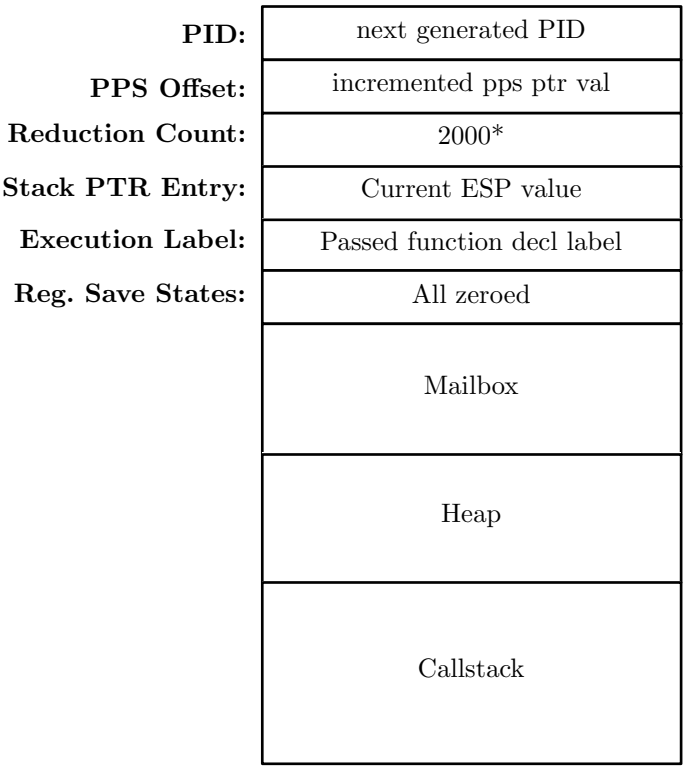


Figure 7.7: A newly initialised process, which has been passed the invocation argument f(x,y,z).

One should note that the evaluation of a process's function application returns nothing after a process dies. If a programmer wishes use this function's return value it must be sent as a message to another process before death.

Also note that its space in the *process store* will not be cleared, this is the job of the process store garbage collector.

7.2.3 Sending Messages

Messages in PLang is extremely simple, they consist only of the source's process ID and the 4Byte integer payload seen in figure 7.8. No other message formats are possible. When a process sends a message using the ! (bang) operator the following events take place:

1. The message is constructed (figure 7.8).
2. The *process store* is traversed linearly, searching for the process which matches the specified destination PID. (**Note:** If the destination process cannot be found, a runtime error will be thrown and PLang will halt.
3. The constructed message will be sent to the destination process's mailbox (using the destination process's *put pointer* to decide where to store it in the mailbox.

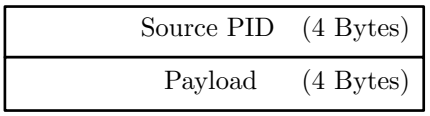


Figure 7.8: A PLang message.

7.2.4 Receiveing Messages

The process mailbox is a queue-like data structure. As shown in figure 7.9, it contains a *put pointer*, a *read pointer* and the messages themselves. The *put pointer* points to the next free space in the mailbox. As described above, this is used when deciding where to store a sent message. Each time a new message is received, this pointer is incremented by 8 Bytes (the size of a message).

The *read pointer* points to the first message in the mailbox which is yet to be read. When a message is read, it is zeroed in the mailbox and the *read pointer* is incremented by 8 Bytes. The *read pointer* will always be pointing

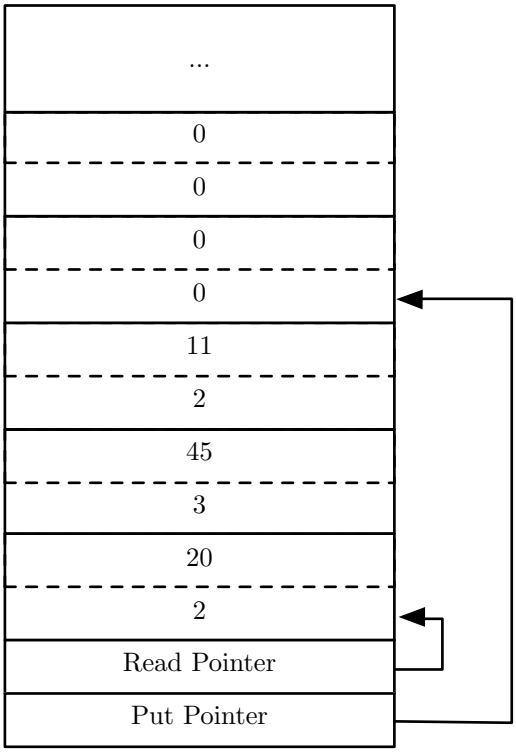


Figure 7.9: A process mailbox.

at the position of the source PID of a message. When PIDs are allocated, they begin with an offset of 1. For this reason, it is possible to use 0 to denote empty messages in the mailbox. Both pointers access the mailbox cyclicly, wrapping around if a boundary is met during incrementation.

7.3 Context Switching

A context switch is used to describe the act of swapping processes in and out of memory during specific intervals in the program. Several design considerations must be made when deciding upon a context switch model.

- At what point in time should a context switch take place?
- What needs to be saved, in order to make sure a process can resume in future without issue?
- How can the program determine the difference between running a process for the first time, and resuming one after a context switch?

The following section explains how PLang approaches and resolves these issues.

7.3.1 Reduction Count

There exists two main methodologies for determining when to context switch: time based switching, or using a reduction count. The former approach works by accessing the system clock in order to interrupt a process after a specific time. A reduction count approach works by initialising a process with a set number of available reductions, e.g 2000. Each function invocation or loop iteration made within a process decrements the reduction count by 1. When the reduction count hits 0 a process is said to be reduced, and a context switch is performed. The latter approach in PLang was chosen for the following reasons:

- This way the language achieves true multi-threading in userspace. A system call to access the clock time requires going through the kernel.
- Using reduction counts is a much simpler model and allows for easy implementation of a priority based scheduling extension.
- Kernel access time incurs a heavy performance hit.

PLang programs are initialised with a default reduction count value of 2000, however, this can adjusted by the programmer at compile time.

7.3.2 Maintaining state

Processes in PLang have access to the `EAX,EBX,ECX` and `ESI` registers which they may use to hold values relevant to a non-finished calculation. For this reason, their state is saved inside the process preamble before they are switched out of memory. The state of the stack and base pointer, `ESP` and `EBP`, are also stored inside the process preamble before context switching.

Note: processes may also use register `EDX` for calculations, but this register is considered volatile and its value in the process preamble is used used to store important information about context switching. For this reason, any contents inside them must be stored on the process call-stack before switching.

7.3.3 Begin or Resume?

The process store may contain newly spawned processes which have not yet begun invocation alongside processes which are part way through execution.

This can be problematic due to the x86 calling behaviour. When a function is called using the call opcode, it returns to the instruction directly beforehand when the instruction pointer hits a return instruction. If this is used incorrectly it will cause a catastrophic break in the program flow. Processes which have not yet been invoked must be called, whereas an unconditional jump should be made to those which need resuming. Clearly, some way of distinguishing between them is required. This is achieved by using the process save state of the `EDI` register. When context switching, if this value is 1 then the process should be resumed (jumped into) otherwise, a 0 indicates it has not yet been started, and instead the call opcode should be used.

7.4 The Function Layout

This section describes how functions are laid out on the process call stack and evaluated in PLang.

7.4.1 Function Calls in PLang

All functions in PLang follow the x86 *cdecl* calling convention. In some cases PLang will call function declarations from the C library, such as printf, malloc and memset. These functions must be called using the cdecl convention and so in order to maintain consistency throughout the language this was imposed on all PLang function calls. Using this convention, function calls which are made from within processes in PLang all take place on the process call-stack, and are handled in the following way:

1. The size of the stack frame is calculated and pushed to the stack.
2. In reverse order, for each argument, a 4 Byte pointer of 1 or 0 is pushed to the stack to correspond to whether it is a pointer or immediate value.
3. The actual arguments are pushed to the stack in reverse order.
4. The return address is pushed to stack. (This is the address of the next instruction to be executed after the function call has completed)
5. The current value of the base pointer is pushed to the stack. (`EBP`)
6. The base pointer (`EBP`) is set to the value of the stack pointer (`ESP`).
7. The body of the function is evaluated, where arguments are accessed via an offset to their position on the stack frame from the base pointer.
8. The functions return value is placed in the accumulator register (`EAX`). In some cases, where a pair of values are returned, register `EBX` is used to hold the second value.
9. The return address stored on the stack is used to return control to the caller.
10. The caller cleans up the stack by adding the stack frame size to the stack pointer.

Figure 7.10 shows a typical stack-frame layout in PLang. The pointer-or-immediate blocks for each argument exist to aid the internal process garbage collector.

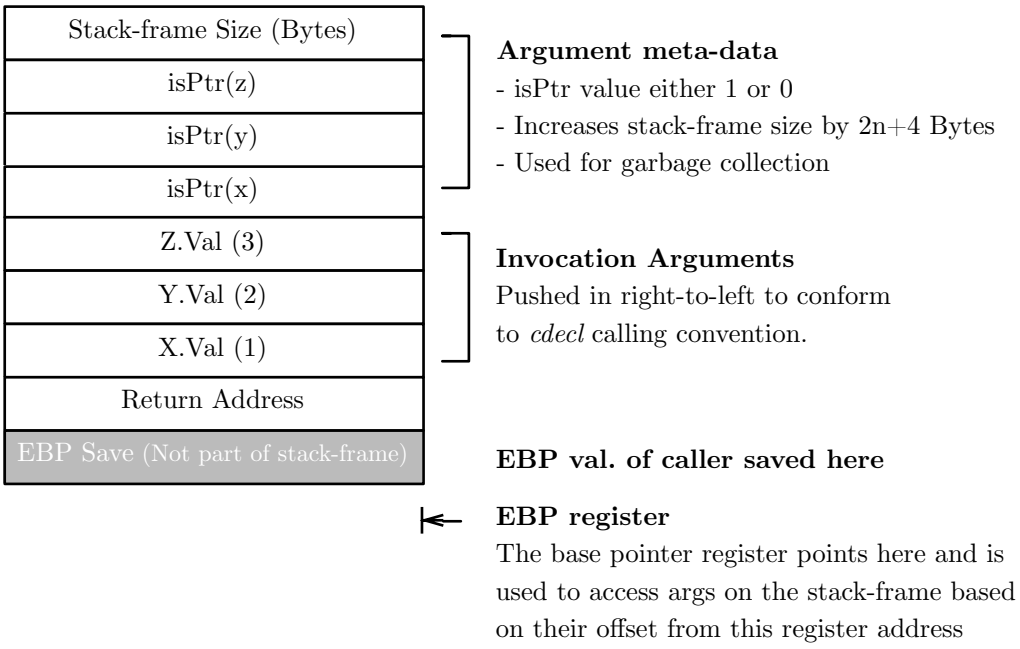


Figure 7.10: A PLang stackframe.

7.4.2 Higher Order Functions

As explained in the front-end compiler documentation, part of the conversion ASTs undergo is nested function flattening. The purpose of this is simplify code generation for higher order functions. With all functions appearing as top level functions, much of the code for generation can be completed with slight alterations to function invocation generation.

Returning a Higher Order Function

Functions are returned as a pair which includes a pointer to the declaration label (placed in the accumulator, `EAX`) and the closure position on the process heap(`EBX`). Figure 7.11 shows the layout of a closure on the process heap.

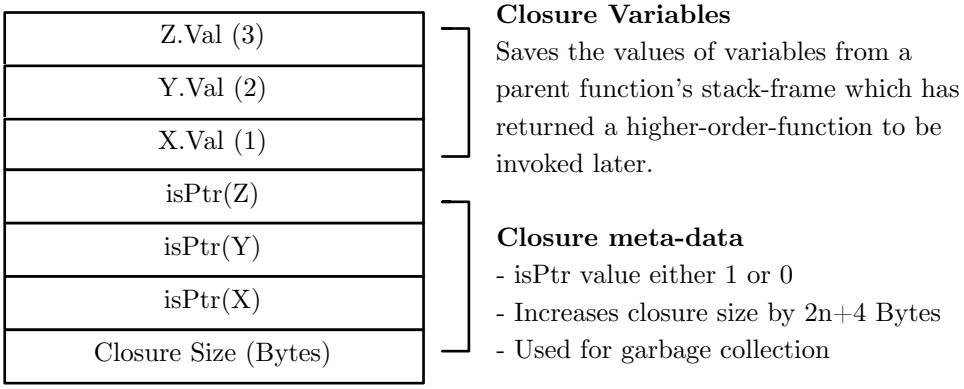


Figure 7.11: A PLang closure.

Calling a Higher Order Function

During the code generation phase of compilation, the nesting level information of the variables inside `g_f` is used to determine where to locate them in memory. Figure 7.12 shows this in detail. When a higher order function is invoked, the pointer to its closure is pushed to its stack frame to allow access to variables which exist in parent functions.

Passing a Function as an Argument

Functions on the stackframe as arguments can be considered a pair, thus requiring 8 Bytes of memory instead of 4 (variables and immediate values).

The nature of the PLang code generator should allow this to be possible without further modification, however, this behaviour is currently unstable and does not work in all cases. Further testing is required before this feature is fully functional.

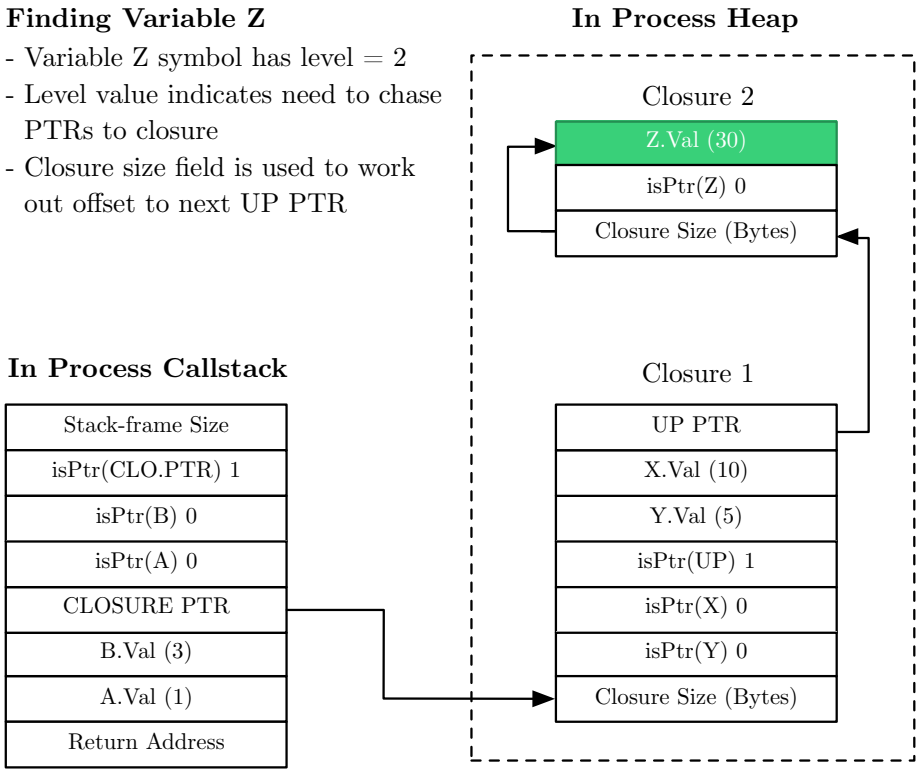


Figure 7.12: Determining a variables location by its nesting level.

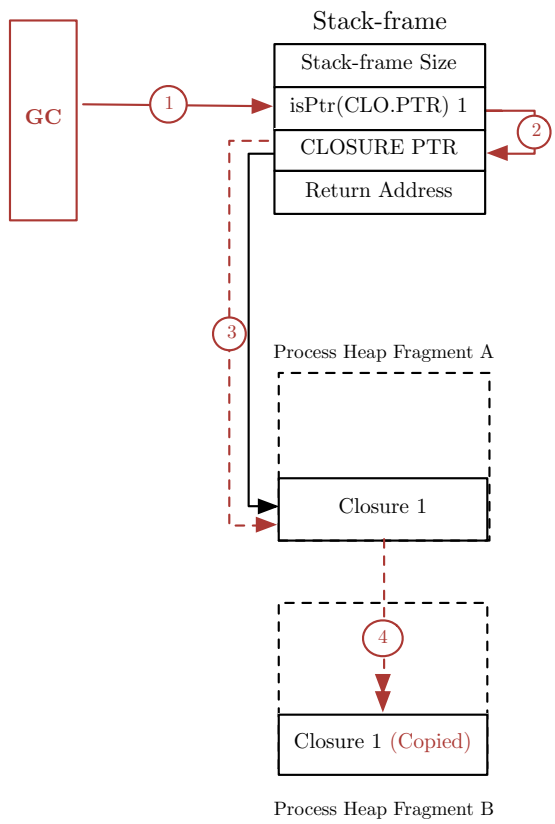
7.5 Garbage Collection

There is no manual memory management in PLang, all heap memory is garbage collected by the program. Each process's heap is individually garbage collected when full, in addition to a main process garbage collector which cleans up processes from the process store after they have died. It is possible that process heaps may contain data that is cyclically referenced, for this reason, the internal process garbage collection relies on the stop and copy algorithm. In contrast, the main process garbage collector can use a more simple reference counted implementation as the PPS contains no cyclic process references. This section will explain how PLang implements both methods in detail.

7.5.1 Internal Process Garbage Collection

In order to preserve the lightweight process model, a process's heap is relatively small. For this reason, garbage collection is necessary in order to prevent heap overflow. References to memory may exist in the process call-stack or the heap itself. This may incur cyclic references which do not work with a simple reference counted garbage collection. The internal process garbage collection uses the stop and copy algorithm. Consider figure 7.13 which shows the following series of events which take place during process heap garbage collection:

1. The current heap fragment has been filled, triggering a GC cycle.
2. The call-stack is traversed, followed by the current heap fragment (A), searching for pointers to data in memory.
3. During this traversal, if a pointer to some heap memory is found, it is copied to the alternate heap fragment (B).
4. Fragment A is then zeroed by the C memset function.



Process Heap GC

1. For each stack-frame on call-stack, search arg. meta-data for isPTR = 1.

2. If isPTR = 1, move to corresponding arg.

3. Get arg PTR val.

4. Copy closure being pointed to to alternate heap fragment.

5. Repeat

After the whole process call-stack has been traversed, this same method is applied to closures in the current heap fragment.

Figure 7.13: Process heap garbage collection cycle.

Reference or Immediate Value

The description of a process layout introduce stack-frames and closures which contain meta-data. In both cases, the size of the addition meta-data is `2n+4 Bytes`, where `n` is the number of arguments. Consider the stack-frame in figure 7.14, it is not possible to tell at run-time which argument is a reference, and which is an immediate value.

The order of the meta-data aligns with each argument in the stack-frame,

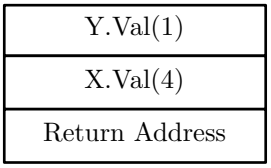


Figure 7.14: A stack-frame with no argument meta-data.

allowing a 1 or 0 block to determine whether an argument is a reference or immediate value respectively. The size of the stack-frame must be the first element of the stack-frame, allowing the garbage collector to know when the meta-data ends and the actual arguments begin. This also allows stack-frames to be delimited by the collection algorithm. Figure **??** shows how the process heap garbage collector uses this meta-data.

7.5.2 Process Store Garbage Collection

The process store is reference counted, allowing processes to be removed from the process heap the moment they are no longer used. As previously explained, this system of garbage collection much simpler as there are no cyclic references so the reference counting model can be implemented. In addition, this is beneficial as there is no long pause in program execution as a garbage collection cycle takes place. Process store garbage collection works as follows:

1. Once a process has completed its execution, the reference to its position on the process store is followed.
2. The given position is zeroed using the C memset library function.
3. The process mapping inside the PPS is also zeroed.

This implementation of reference counting is extremely simple, as a process will only consist of at most one reference. This means the moment that a process is killed, it is cleared from the process heap. This process is shown visually in figure 7.15.

When Current Process Dies
(Process GC by Reference Counting)

1. The current process pointer is followed from PPS value

2. The current process pointer mapping inside the PPS is followed to the process position on the process store.

3. The process frame in the process store is zeroed. (using memset)

4. The process mapping inside the PPS is zeroed

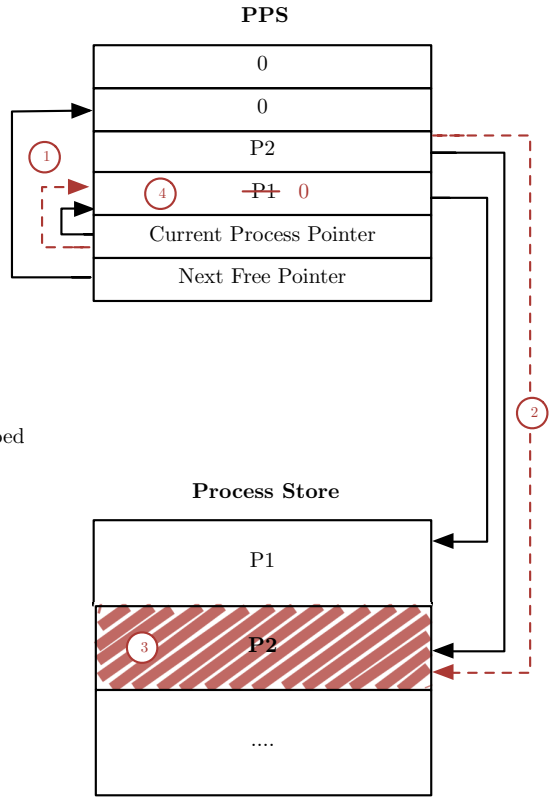


Figure 7.15: Removal of process from process heap after death.

Chapter 8

Testing

The correctness of the compiler was checked via unit testing from the ScalaTest library. This was run directly from SBT (Scala Build Tool) while in test mode. The front-end and back-end of the compiler was tested together with the logic that should an aspect of compilation fail, the results would be catastrophic. With this in mind, during the build process, each part of the compiler pipeline could be checked by hand to see whether it was responsible for the failure. For each test, the following actions took place:

- A source program was provided which, when run, would print some output to the stdout.
- The source program was compiled using the PLang compiler.
- The PLang program was copied to a Linux Ubuntu virtual machine via SCP.
- A bash script on the virtual machine ran the program, and copied the output back to the machine running the compiler and test-suite.
- The output was then compared with an expected value, which was hard-coded into the test.

Concurrency was tested by having each process print its invoked functions return value. These were then cross-referenced against an expected list of processes with their expected print values where order was not important.

A total of 40 tests were ran, including tests on inputs which should fail.

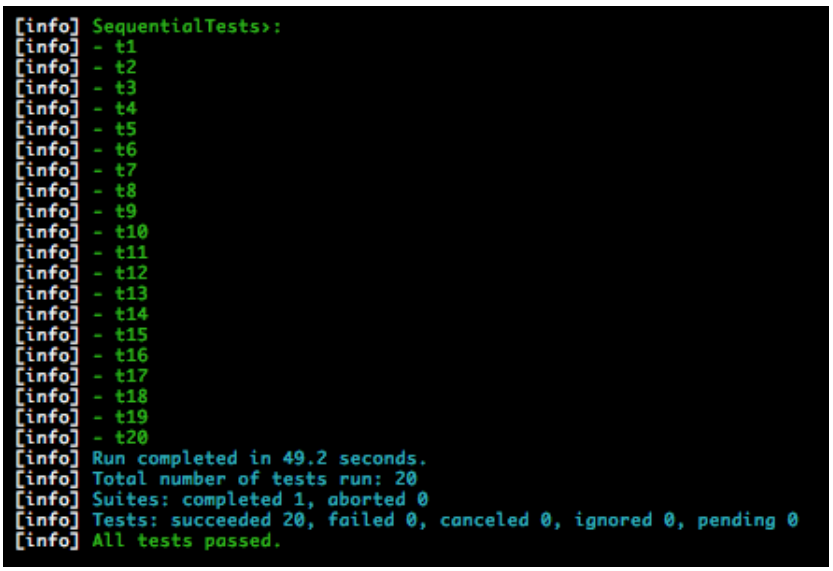


Figure 8.1: Tests on sequential components of the compiler only.

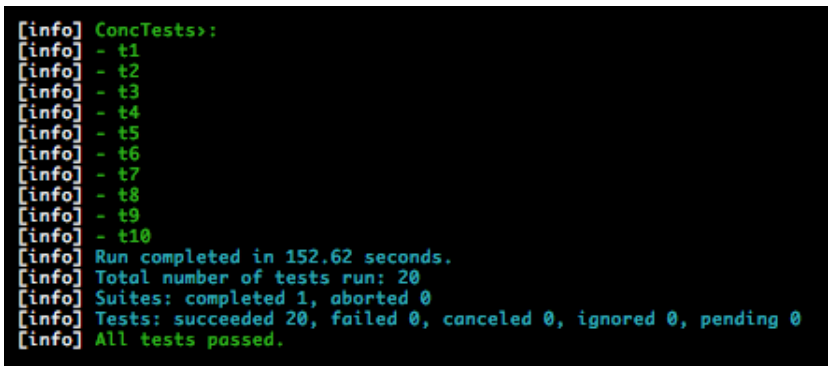


Figure 8.2: Tests on concurrent components of the compiler.

Figure 8.3: Tests on inputs which should fail.

Chapter 9

Evaluation

This chapter aims to evaluate PLang and assess how well it reached the aims it set out to achieve.

9.1 Compiler Correctness Evaluation

The unit testing of the PLang compiler helped to ensure that many of implemented features ran as expected. However, testing concurrency is a difficult task and it requires much effort beyond the scope of this project in order to be satisfactory to meet the standards required for industrial compilers. This is because there are many possible branches of execution in concurrent programs. The current compiler test-suite has several flaws that would need to be fixed before being sufficient for an industrial compiler:

- The set of tests was too small.
- The order in which concurrent processes were executed was not measured. This meant that it had to be reasoned about by hand, leading to possible human error and made it difficult to reason about execution on concurrent programs with many processes.
- It was not possible to inspect the state of the program stack or registers at specific parts of runtime. This was achieved by printing them to the stdout, this could introduce errors and not be completely accurate.

9.2 Performance Evaluation

In addition to compiler correctness testing, this project attempted to test the performance of PLang's runtime using benchmark tests. The following C functions were embedded into the assembly code at various points within a PLang programs upon launch:

```
int beginBenchmark() {
        startTime = (float)clock()/CLOCKS_PER_SEC;
}

int endBenchmark() {
        endTime = (float)clock()/CLOCKS_PER_SEC;
}

int getResult() {
        return (endTime - startTime);

}
```

Unfortunately, it was learned during this stage that testing performance speed of x86 assembly language at runtime is non-trivial and many variables must be eliminated to achieve any significant results. This testing stage was not able to produce any meaningful results for the following reasons:

- The C benchmarking function was called from assembly, and it was not known how to correctly isolate this to specific components in the program such as process spawning or context switching due to their complexity.
- It was possible that garbage collection cycles interrupted process spawning which would produce an unreliable result.
- Operating system functions running within the linux environment could have impacted upon the benchmark result.
- Many of the benchmarking tests resulted in segmentation faults which suggests that the state of the stack required close inspection to realign it before and after the benchmark function calls.

- The PLang programs were run in a virtual machine which was assigned variable memory based on usage. Benchmarking may have been affected by the way that the virtual machine handles this.

With additional time and research, this report would aim to resolve these issues by understanding how existing concurrent programming languages are benchmark tested. The testing of two main performance aspects of PLang were attempted: The degradation of process spawn time over the course of the runtime, and the comparison of different reduction counters to consider the tradeoff between throughput and latency masking.

9.3 Implementation Evaluation

The implementation of PLang overall was successful and much was achieved given the time constraints. There are, however, several weaknesses in the current version of the compiler:

Inefficient Use of Memory Space

The meta-data used in process stack-frames and closures would need to be reconsidered before deploying PLang as an industrial strength compiler. Doubling the size of a stack-frame and closure uses an unacceptably wasteful amount of memory and a different method of differentiating between pointers and immediate values would need to be considered. One interesting approach to this would be to investigate whether removing garbage collection altogether, and managing memory through tracking ownership of data statically, similar to the Rust programming language.[6]

Stack Offset Value Oversight

When building the PLang compiler, many elements, such as the message passing, garbage collection and function generation were built modularly so as to conform to the open/closed principle of software engineering. However, when designing the main program stack, the offset values were often hard coded into the assembly language environment. This meant that adding or

removing unnecessary values would be incredibly difficult as it would affect the offset of all other values. Part way through the development of PLang, it was discovered that another field was required in the process preamble to determine whether the process had started its execution. In order to resolve this with the least disruption to development as possible, the `EDX` register's save position on the process was used to house this value.

9.3.1 Scheduling Issues

The current PLang compiler adds newly spawned processes to the next free (zero) position on the process pointer scheduler (PPS). It begins at the current process and increments the PPS until it finds the next zeroed address inside the PPS. This means that new processes are not necessarily added to the end of the scheduler in true round robin fashion and may be inserted before the turn of other processes.

9.4 Project Plan Evaluation

For the most part, the project plan was stuck to and successful. Initially, process message sending was intended to be achieved via channels from the pi-calculus model. However, due to time constraints, a simpler implementation of the mailbox approach was carried out instead. The original plan set March and April for draft plan and dissertation, however, bug fixing on the main implementation meant that this process was slightly delayed. In addition, fixing errors implementation meant that less time was possible to carry out performance testing, which was found to be more challenging than expected.

Chapter 10

Conclusion

This project has provided an insight into the issues surrounding concurrency and provided a alternate programming language prototype to the common shared memory approaches familiar to many programmers.

The main objectives of this report have been achieved and the simplicity of the source language makes it ideal for the target audience. The data from this report has provided an insight into the various methods used to approach the issue of concurrency as well as the complexities of designing a program which handles concurrency in assembly language. It also shows that properly testing a concurrent system is challenge in itself and one could argue that it is still an unsolved issue.

There are several future improvements that PLang could benefit from. Listed below are a few of the most important additions which will hopefully make their way into future developments:

- Runtime error catching
- Extension of base types to include collections such as lists, sequences, maps and arrays.
- Inclusion of functional combinators (such as map, fold, filter, group, zip) as standard library functions.
- Switch of mailbox message sending design to the process calculi model.
- Priority scheduling of processes
- Usage on multi-core environment
- Pattern matching
- Dynamically linked stacks which start small and can grow arbitrarily large.

- Improved error messages in parser and type-checker

Moving forward, PLang will remain publicly available on Github and continue to be extended and improved upon. It will be free for people to contribute to and modify in the hopes that programming language enthusiasts will see it as an exciting and enjoyable project to develop.

Appendix A

The PLang Context-Free Grammer

```
prog          ::=   funclist
funclist      ::=    ϵ | func funclist
func          ::=   DEF id LPAREN argdec RPAREN EQ block

argdec        ::=   id argdecmore | ϵ
argdecmore    ::=   ϵ | argdecmore COMMA id

block         ::=   LBRACE sequence RBRACE
sequence      ::=   e | e SEMICOLON sequence

e             ::=   WHILE LPAREN e comp e RPAREN block
                    | func
                    | IF LPAREN e comp e RPAREN block ELSE block
                    | id
                    | id EQ e
                    | block
                    | id LPAREN args RPAREN
                    | LPAREN e RPAREN
                    | e binop e
                    | e comp e
                    | PRINT LPAREN e RPAREN
                    | RECEIVE
                    | e
                    | SKIP
                    | SPAWN LPAREN e RPAREN
                    | e BANG e
                    | integer
                    | bool
```

```
args    ::=     e argsmore | ϵ
argsmore ::=    ϵ | argsmore COMMA e

id      ::=     IDENTIFIER

comp    ::=     EQEQ | GTEQ | LTEQ | GT | LT

binop   ::=     PLUS | MINUS

integer ::=     INTEGER_LITERAL(n)

bool    ::=     BOOLEAN_LITERAL(b)
```

Appendix B

The AST Classes

```
sealed trait AST

case class Program(funcs:List[Function]) extends AST

case class Function(name:String, args:List[Exp], body:Exp) extends
   Exp

case class Thread(name:String, body:Exp)

abstract class Exp extends AST

case class Assign(name:String, var offset:Int, e:Exp) extends Exp

case class Binexp(l:Exp, op:Binop, r:Exp) extends Exp

case class Bincomp(l:Exp,comp:Comp,r:Exp) extends Exp

case class BoolLiteral(b:Boolean) extends Exp

case class IntLiteral(n:Int) extends Exp

case class If(l:Exp, comp:Comp, r:Exp, thenBody:Exp, elseBody:Exp)
   extends Exp

case class While(l:Exp, comp:Comp, r:Exp, body:Exp) extends Exp

case class Invoke(name:String, args:List[Exp]) extends Exp

case class Seq(l:Exp,r:Exp) extends Exp

case class Skip() extends Exp
```

```
case class Spawn(f:Exp) extends Exp

case class Message(dest:Exp,payload:Exp) extends Exp

case class Print(s:Exp) extends Exp

case class Concat(l:Exp,r:Exp) extends Exp

case class InnerFunction(name:String, args:List[Exp], body:Exp)
   extends Exp

case class Functional(name:String, offset:Int, level:Int) extends
   Exp

case class Pair(name:String, offset:Int, level:Int) extends Exp

case class Closure() extends Exp

case class Receive() extends Exp

case class FunctionalPlaceholder() extends Exp

case class Variable(name:String, offset:Int, level:Int) extends Exp

case class VariablePlaceholder() extends Exp

abstract class Comp

case class Equals() extends Comp

case class Greater() extends Comp

case class GreaterEq() extends Comp

case class Less() extends Comp

case class LessEq() extends Comp

abstract class Binop
```

```
case class Plus() extends Binop

case class Minus() extends Binop
```

Appendix C

Libraries Used

Name	Author	Website	Licence
JFlex	Gerwin Klein, Steve Rowe, and Rgis Dcamps.	http://jflex.de/	BSD-style license
CUP	Scott Hudson, Frank Flannery, C. Scott Ananian, Michael Petter	http://www2.cs.tum.edu/projects/cup/index.php	GPL
NASM	The Netwide Assembler Team	http://www.nasm.us/	The BSD 2-Clause License
SBT	TypeSafe	http://www.scala-sbt.org/	GPL

Appendix D

Project Plan

In order to ensure the smooth completion of this project within the given time frame, it must be broken down into phases which need to be completed by various deadlines over the time.

Research Phase This phase covers becoming familiar enough with the topics that they can be successfully implemented according to the current standards. There is a large interdependency in this stage with the implementation. This means that for some research tasks, their expected completion date will be the same as that of their respective implementation task.

- Research Type Inference
- Research x86 Architecture
- Research scheduling in assembly
- Research how closures work in higher order functions
- Research how garbage collection implementations work
- Research how current concurrent programming languages implement thread based message passing
- Research how message passing relates to a type system

Implementation Phase This will be the most time consuming phase of the project, it also presents with the most difficulty in scheduling, as owing to the nature of the compiler pipeline, new implementation tasks cannot be initiated until the completion of the previous task.

- Write a command-line UI for the compiler
- Write a lexer for the sequential language
- Write a parser for the sequential language

- Write a type inference algorithm for the sequential language
- Generate code for the sequential language
- Implement higher order functions into the sequential language
- Amend the sequential language to include terms for concurrency
- Amend the language to support multi-threading
- Add a garbage collector to the language
- Implement message passing between threads

Evaluation Phase The phase also has an interdependency with the implementation phase, as a vast portion of the project testing will be based on test driven development and testing of each stage of the compiler pipeline before moving on to the next one.

- Test each new addition to the compiler pipeline
- Final evaluation of thread spawning and message passing effectiveness

Writing Phase The deadlines for the writing phase are very rough, this process will be undertaken as the project goes along and so it would not be accurate to specific an explicit start date.

- Interim report
- Draft report
- Final report

Appendix E

Meeting Log

30/10/15
Discussion of HOF implementation using closures. Went over drafts and questions about formatting interim report

02/10/15
First discussion about HOF, how the heap can be used for closures and its introduction from Scheme. Also discussed timing of project

22/09/15
Discussion of how to convert sequential language into concurrent language. Also discussed stack alignment issues present in x86.

03/09/15
Discussion of what to include in proposal for FYP, narrowed list down to select few which are feasible within given time frame.

24/01/16
Discussed problems with stop and copy garbage collection technique.

03/02/16
Went through different approaches to determining reference or immediate values in a tracing garbage collection scheme.

12/02/16
Showed working implementation of nested higher order functions and garbage collection.

09/03/16
Discussed problems with putting all elements of scheduling, higher order functions and garbage collection together.

24/03/16
Skype meeting. Discussed problems within the AST converter and

how to change it so that different transformations are performed sequentially.

30/03/16

Errors in PLang were found in the higher order function aspect of compilation, we attempted to find out why and provide a fix for them.

13/04/16

Discussion of draft report and how to improve writing for final submission.

Chapter 11

References

[1] Martin Berger. Week 1 lecture slides: Compilers and computer architecture, sussex university.

[2] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263, 1996.

[3] Francesco Cesarini and Simon Thompson. *Erlang programming.* " O'Reilly Media, Inc.", 2009.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[5] BCS The Charted Institute for IT. The code of conduct, 2015. Accessed: 2016-04-15. URL: `http://www.bcs.org/upload/pdf/conduct.pdf`.

[6] Mozilla Foundation. The rust programming language, 2010. Accessed: 2016-04-14. URL: `https://www.rust-lang.org/`.

[7] LLVM Developer Group. LLVM low level virtual machine, 2003. Accessed: 2015-09-30. URL: `http://llvm.org`.

[8] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[10] Edmondo Pentangelo. Hindley milner type inference sample implementation, 2010. Accessed: 2016-04-06. URL: `http://fsharpcode.blogspot.co.uk/2010/08/hindley-milner-type-inference-sample.html`.

[11] Kevin Poulsen. Software bug contributed to blackout, 2004. Accessed: 2016-04-06. URL: `http://www.securityfocus.com/news/8016`.

[12] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[13] Bruce Schneier. Race condition exploit in starbucks gift cards, 2015. Accessed: 2016-04-06. URL: `https://www.schneier.com/blog/archives/2015/05/race_condition_.html`.

[14] Christopher Strachey. Understanding programming languages. *Higher-Order and Symbolic Computation*, 13, 2000.

[15] Andrew S Tanenbaum. *Modern operating systems*. Pearson Education, 2009.

[16] A. M. Turing. Computability and -definability. *The Journal of Symbolic Logic*, 2:153–163, 12 1937. Accessed: 2016-04-06. URL: `http://journals.cambridge.org/article_S002248120004072X`.