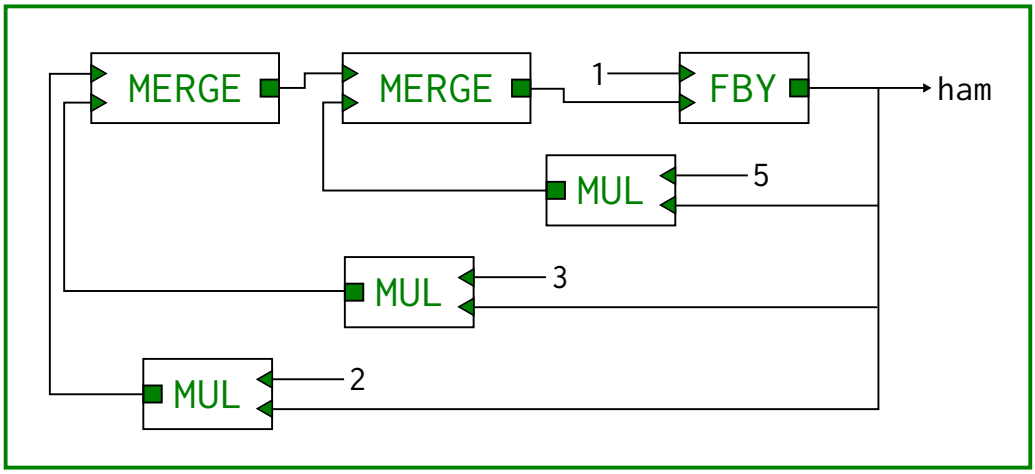
Jonas Kaiser



Reconsidering Lucid

a modern approach

Computer Science Part II

Churchill College

May 12, 2010

The cover diagram shows a dataflow graph which computes the Hamming sequence. These numbers are also known as *5-smooth*, i.e. they have no prime divisors greater than 5.

Proforma

Name:	**Jonas Kaiser**
College:	**Churchill College**
Project Title:	**Reconsidering Lucid – a modern approach**
Examination:	**Computer Science Part II, May 2010**
Word Count:	**ca. 11700**[1] (to the nearest 100)
Project Originator:	Jonas Kaiser
Supervisor:	Dominic Orchard

Original Aims of the Project

To implement a modern, high-level interpreter for a subset of the dataflow programming language Lucid. The implementation language of choice is Scala due to its actor based concurrency model.

Work Completed

The simLucid interpreter as required by the proposal, plus two proposed extensions (caching and a timing evaluation) and one extension that was not initially planned (going from one- to multi-dimensionality, using declarations).

Special Difficulties

Eyjafjallajökull

[1] This word count was computed by `detex <texfiles> | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Jonas Kaiser of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to Dominic Orchard who supervised this project and supported me with extensive feedback and the ability to use his machine for the core-scaling evaluation.

Chapter 1

Introduction

For my project I successfully developed a parallel interpreter for the dataflow language Lucid, called *simLucid*. simLucid implements a core subset of the Lucid language using the high-level programming language Scala. Its correctness is demonstrated by comparing the implementation against the existing pLucid interpreter, as well as comparing against operational semantics outlined in various publications. This satisfies the core requirements of my project proposal. Furthermore three extensions, namely a simple caching system, extension to multi-dimensionality, and a performance evaluation have been developed.

1.1 Background and Motivation

At the end of the last century, single-core processor designs hit a wall of diminishing performance returns due to engineering and physical constraints. To keep up with Moore's law, manufacturers began to design dual core and later quad core architectures. Processors with higher numbers of cores are becoming predominant, with heterogeneous designs also appearing [9]. Each separate core in these many-core machines tends to be weaker than their single-core counterparts, primarily to reduce power consumption.

High performance on many-core systems can be achieved through parallel programming which matches the underlying architecture. Most mainstream imperative languages are grounded in single-core von Neumann architectures, designed to deal with a single thread of execution. The parallelism requirement prompted the addition of threads as new primitives. However, threads introduced trouble into the well-behaved world of sequential execution [13]. The most severe prob-

lem with this approach is that parallelism has to be expressed and dealt with explicitly by the programmer.

Firstly, this dramatically increases the complexity of the programming task. Secondly, it can introduce errors which arise from non-deterministic interaction. Such bugs are hard, or even impossible, to cover with exhaustive testing. Thirdly, the degree of achievable parallelism using existing tools seems to be limited. So far it has been difficult to demonstrate reasonable gains beyond four cores for wide sets of applications [4, 18, 14].

The obvious question then is: *What can we do better?* One option is a paradigm shift from imperative to declarative languages. The latter tend to hide possible parallel execution of programs in their semantics and implement it in a runtime system, taking that burden away from the programmer. In many declarative programming languages, parallelism may be possible because of the absence of side-effects which gives freedom in the order of evaluation. Apart from functional (ML, Haskell) and logic (Prolog) languages there is a further member in this class which is often overlooked: dataflow languages.

Dataflow languages expose parallelism by removing unnecessarily strict control flow constructs where they are not needed to express the essence of an algorithm or data structure. A program written in one of these languages can be interpreted as a network of processes, with data tokens travelling along channels and nodes performing computations [11]. One such language is Lucid [19].

If we consider the growing number of cores on a chip it might become feasible to dynamically rewire a chip to form a dataflow network as an execution strategy. To investigate this option, we need a framework which is modular and flexible, that can ideally mimic various many-core designs and accept a generic dataflow network representation of a program.

The only Lucid implementation to date is pLucid [6, 7], which is unfortunately unsuitable for the proposed approach. pLucid, albeit fast, was not developed with extension or parallelism in mind. The source code is written in low-level C with complicated optimisations. Each compiler stage sits in a separate executable with significant code duplication and subtle dependencies. Additionally, documentation is next to non-existent. Furthermore, pLucid forces the dataflow language into the sequential von Neumann world using concepts of data warehousing and *eduction* [19, 7]. Another issue was revealed during evaluation: for some programs pLucid outputs incorrect results (details in Chapter 4).

In this project, I propose to take a first step in the direction of the framework

suggested by building a new interpreter that honours the dataflow nature of the language with a back-end that is based on parallel dataflow networks. The eduction concept is maintained but the single warehouse of pLucid is distributed over the set of computation nodes to allow parallel execution.

The simLucid implementation is far from a full many-core simulator, but it serves as a proof of concept for this investigation, and a stepping stone for future research.

Chapter 2

Preparation

2.1 Getting started

2.1.1 Notation

Throughout this dissertation, all Lucid code is typeset as

```
X = 0 fby X + 1
```

and Scala code is presented as

```
Console.println("Hello, World!")
```

Inline typesetting is similar: `Lucid` and `Scala`.

2.1.2 Language Research

Before I could start I had to acquaint myself with both the target language, Lucid, and the intended implementation language, Scala. For Lucid, this involved comprehending a lot of (fairly old) research literature. To get a feel for the language I obtained a copy of the pLucid system and experimented with the example programs illustrated in the well-known *Lucid Book* [19]. Sources on multi-dimensional programming [2], early publications on the language specification [3] and a presentation of modern trends in the field [16] were also very helpful

to grasp the underlying principles. I also came across GLU [10], which takes Lucid as a metalanguage to join together fragments of C code in a dataflow oriented way. This was not directly relevant to this project but helped to establish the wider picture and gives credence to further parallel investigations.

For Scala the case was slightly different. I had worked with the language before on small projects and hence was aware of, but not proficient with, the actor concurrency library and the parser combinator framework. Hence, most of my preparation went into writing a command line calculator using parser combinators and the development of a standalone, actor-based implementation of the *howfar* Lucid program (see Chapter 4 for details). The latter confirmed the suitability of my ideas for the back-end of simLucid.

2.1.3 Work Environment

I used my home machine for most parts of the development, while data was kept in an svn repository located in my PWF home directory for backup. Since most Scala IDE plugins still occasionally exhibit major glitches I decided to work in a shell-only environment (editor: `vim`). In retrospect I would opt for a different code management system (probably git[1]) as the shell interface of svn is at times cumbersome to use, especially if something goes wrong in the directory tree. For diagrams I decided to use the open source vector graphics program Inkscape[2] due to its wide range of image formats. For performance curves I used `gnuplot`. Along the way I learned a fair amount of shell scripting and learnt the use of `sed` and `dc` to simplify the evaluation process.

2.1.4 Methodology

At the beginning I was not sure what amount of Lucid's language specification I would be able to complete, hence I decided to adopt an iterative development methodology (the *Spiral Model*). I started with as minimal a grammar as possible for a working front-end and linked it to a very early version of a network based back-end. After that I could add more features, alternating between these two major aspects of the system. Overall that approach worked reasonably well. At two points I had to perform a large-scale refactoring of the system layout due to difficulties in accommodating particular features. Such reorganisations throttled

[1] `http://git-scm.com`
[2] `http://www.inkscape.org`

progress somewhat, but the effect was not too severe, thanks to well organised code. To ensure that newly added features did not break the work achieved so far I set up a sequence of integration tests based on sample Lucid programs of increasing feature richness.

2.1.5 Success Criteria

I defined the requirement for success to be a working simLucid interpreter with a semantically correct front- and back-end (Appendix D). The system should implement a first order subset of the Lucid language specification with integers and the two basic, but fundamental, intensional operators (see below). Multi-dimensionality using declarations is not required, but was added as an extension.

2.2 The Dataflow Language Lucid

2.2.1 How to make Iteration Declarative

Lucid is a pure, definitional language with lazy evaluation. The main design objective was to preserve the concept of iteration whilst still being declarative.

A crucial concept used in Lucid is the distinction between *extensions* which are ordinary values and *intensions* which are maps from *contexts* to *extensions*.

$$\text{intension} : \text{context} \rightarrow \text{extension}$$

Such a map models entities which have a *context*-dependent meaning. In the case of iterative programs one can observe that a single variable can take on multiple values and whenever the variable is used, its value (or *extension*) depends on the point in the dynamic execution trace, i.e. variables are *intensional* entities. A comprehensive presentation of the concept of *intensionality* is given in [17].

A dynamic execution trace is a linear sequence of instructions: so, without loss of generality, we can assign each position a discrete "time" context, represented using natural numbers.

The designers of Lucid made this intensional nature of variables explicit by creating an algebra of *histories*. A *history* is a map from natural numbers to elements of a different algebra. A *history* is conceptually identical to an infinite stream.

We take an existing algebra where the elements are integers, reals, booleans, etc. and use a stream functor to give an algebra where the elements are streams of integers, reals, booleans, etc. Literals from the underlying algebra are lifted to streams that have the given *extension* at every context, i.e. constant streams that will be denoted syntactically by the same literal. Finally, operators from the base algebra are lifted to operators of the same arity which work pointwise on streams. Operators generated in this way are *extensional*, i.e. they produce result streams whose *extension* at a particular timestamp (*context*) t only depends on the *extensions* of all operands at the same time t. Operators with this property are also called *synchronic* or *pointwise* in previous Lucid literature [19].

In Lucid the set of *extensional* operators is augmented with two *intensional* operators. Their result at time t can depend on the whole (or partial) *history* of an operand. They give sufficient control over the relative positioning of streams without exposing time indices directly as first class values.

2.2.2 From Algebra to Programming Language

Lucid is a member of Landin's ISWIM family of languages [12] parameterised by the continuous algebra we have constructed above. Lucid does not use the full ISWIM framework and makes the following changes: only the first order subset of ISWIM is used (functions are not first class values). Also ISWIM distinguishes *where clauses* that do not allow for mutual recursion (introduced using the **`where`** keyword) and those which do (**`whererec`**). In Lucid all *where clauses* can contain mutually recursive definitions, and for brevity only the **`where`** keyword is used.

2.2.3 Operators and Operands

In Lucid all operands, constants and variables alike, denote infinite streams of values. Accordingly all operators produce new streams. Here are some examples.

$$\begin{aligned} \texttt{42} &= \langle 42, 42, 42, 42, 42, \ldots \rangle \\ \texttt{x} &= \langle x_0, x_1, x_2, x_3, x_4, \ldots \rangle \\ \texttt{index} &= \langle 0, 1, 2, 3, 4, \ldots \rangle \end{aligned}$$

Lucid's *extensional* operators work pointwise,

$$\begin{aligned} \texttt{a} &= \langle a_0, a_1, a_2, \ldots \rangle \\ \texttt{b} &= \langle b_0, b_1, b_2, \ldots \rangle \\ \texttt{a OP b} &= \langle a_0 \text{ op } b_0, a_1 \text{ op } b_1, a_2 \text{ op } b_2, \ldots \rangle \end{aligned}$$

where **OP** and "op" are the lifted and base versions respectively of operators such as addition, multiplication, modulo, equality or less than. A conditional *if-then-else-fi* construct is treated as such an operator, albeit a ternary one.

The two core *intensional* operators mentioned above are the binary **fby** (pronounced 'followed by') and the unary **next** which conceptually allow shifting a sequence one step to the right or the left respectively. While the latter drops the leading element, the former must fill the empty slot at time 0 and hence requires two operands:

$$\begin{aligned} \texttt{x} &= \langle x_0, x_1, x_2, \ldots \rangle \\ \texttt{y} &= \langle y_0, y_1, y_2, \ldots \rangle \\ \texttt{x fby y} &= \langle x_0, y_0, y_1, y_2, \ldots \rangle \\ \texttt{next x} &= \langle x_1, x_2, x_3, \ldots \rangle \end{aligned}$$

There are several other useful *intensional* operators that can be derived from **fby** and **next** using function definitions.[3]

The unary **first** operator returns a constant stream that has everywhere the extension its argument has at time 0. The binary **asa** (short for 'as soon as') works similarly, but the value picked depends on the time when the guard (the second operand) is **true** for the first time. The remaining two are **whenever** (sometimes aliased as **wvr**) which is the sequence of only those values from the first operand for which the corresponding guard value is **true** (i.e. a simple filter) and **upon** which delays its first operand and only advances it when the guard is **true**, otherwise the previous value is repeated.

$$\begin{aligned} \texttt{x} &= \langle x_0, x_1, x_2, x_3, x_4, \ldots \rangle \\ \texttt{g} &= \langle F, T, T, F, T, F, F, T, F \ldots \rangle \\ \texttt{x upon g} &= \langle x_0, x_0, x_1, x_2, x_2, x_3, x_3, x_3, x_4, x_4, \ldots \rangle \end{aligned}$$

[3]However since recursive functions tend to be a bad choice in a dataflow environment, these are normally implemented directly, using recursive stream equations.

2.2.4 Laziness

To highlight the difference between lazy and eager semantics consider this example:

```
next X
  where
    X = X fby 1;
  end
```

Here `X` and `next X` have different denotations depending on whether the semantics are eager or lazy:

$$[\![\texttt{X}]\!]_{\text{eager}} = \bot \qquad [\![\texttt{X}]\!]_{\text{lazy}} = \langle \bot, 1, 1, 1, \ldots \rangle$$
$$[\![\texttt{next X}]\!]_{\text{eager}} = \bot \qquad [\![\texttt{next X}]\!]_{\text{lazy}} = \langle 1, 1, 1, \ldots \rangle$$

A lazy evaluator will never ask for `X` at time 0, so the diverging case is not exposed, as required by Lucid's semantics.

2.2.5 Definitions and Declarations

The body of a *where clause* consists of a set of definitions. Definitions are the backbone of Lucid programs. They express intensional equality, i.e. pointwise equality between sequences. The *lhs* of such an equation can either be a variable name or a function symbol with formal parameters, while the *rhs* can be an arbitrary Lucid term.

From a declarative point of view this method specifies, in a single equation, a property that will always be true for a particular variable. From an operational perspective they give a recipe to compute the value for a particular timestamp. Consider the following definition of the inbuilt sequence `index` that strongly resembles the inductive mathematical definition of natural numbers:

```
index = 0 fby index + 1
```

There are a few definitions which, albeit being well formed, will not yield any useful computational result, since any demand will cause the program to diverge. The following is an example of such a definition:

```
x = next x
```

Its denotation is a stream of $\perp$s. One might say that this is obviously ill-defined since `x` is expressed in terms of its own future but this statement is not true in general and leads to one of the more interesting features of Lucid. Chapter 4 will show how one can make use of such *future references*.

With the constructs presented so far, all streams vary in one dimension only. There are, however, many useful algorithms which rely on a notion of subcomputation, i.e. for each step in an outer iteration there is a whole sequence of inner iterations. In Lucid this is accomplished with a construction called a *declaration* of the form:

```
X is current y
```

As soon as *declarations* are present in a clause the internal computations progress in a time dimension nested inside that used by the subject of the clause.[4] Now in this inner time dimension `X` is a constant sequence, having everywhere the extension `y` had at the current outer time dimension.

$$\begin{aligned} \texttt{y} &= \langle y_0, y_1, y_2, \ldots \rangle \\ \texttt{X is current y} &= \langle \langle y_0, y_0, y_0, \ldots \rangle, \\ &\quad \langle y_1, y_1, y_1, \ldots \rangle, \\ &\quad \langle y_2, y_2, y_2, \ldots \rangle, \\ &\quad \ldots \rangle \end{aligned}$$

Thus `is current` provides streams of streams, or equivalently streams varying in more than one dimension. See Chapter 4 for an illustration of this feature in calculating factorials.

2.2.6 Input and Output

A well-formed Lucid program consists of a top-level term, which might be a simple expression or a *where clause* with nested terms. The values of the stream generated by this top-level term are considered as the output of the program.

A term may use variables which are not defined in its scope. These variables are called the *globals* of the term and are considered to be inputs of the program. These variables are globally unique, i.e. multiple occurrences refer to the same stream of input values. Input requests are directed to the user.

[4] Note that this is a non-local effect on the meaning of the whole program, with all complications that this may entail.

2.2.7 Example – The Hamming Problem

Dijkstra attributes the problem of generating all numbers of the form $(2^i 3^j 5^k | i, j, k \geq 0)$ in ascending order and without duplication to Hamming [5]. It is commonly used to demonstrate the power of lazy and/or functional languages based on how gracefully they deal with infinite data structures. Imperative implementations on the other hand usually cannot match such elegance.

He observed that if H is the sequence we are looking for, then $H*2$, $H*3$ and $H*5$ (using pointwise multiplication) are all subsequences, and their ordered, duplicate-free merge recreates H, apart from the leading element: 1.

This description can be translated into the following Lucid code:

```
h
  where
    h = 1 fby merge(merge(2 * h, 3 * h), 5 * h);
    merge(x, y) = ...
  end
```

The **merge** function is assumed to maintain the invariant that if both operands are sorted in ascending order and they are duplicate-free then so is the result of the merge. In Haskell on could write a recursive definition:

```
merge :: [Int] -> [Int] -> [Int]
merge     []     ys = ys
merge     xs     [] = xs
merge (x:xs) (y:ys) = if x < y then x : merge xs y:ys
                      else if x > y then y : merge x:xs ys
                      else x : merge xs ys
```

The use of recursive functions, however, often leads to poor performance in dataflow systems. The following alternative is a better choice.

Given streams X and Y, take values from stream X while they are smaller than the leading value of Y, then take values from Y until they exceed X again and repeat ad infinitum, i.e. one stream is "held up", while the other is consumed.

This yields the following definition of **merge** where the **upon** operator delays the two inputs as appropriate:

```
merge(x, y) = if xx <= yy then xx else yy fi
  where
    xx = x upon xx <= yy;
    yy = y upon yy <= xx;
  end
```

The complete program is used in Chapter 4 for correctness and performance tests and a more detailed discussion can be found in ([19], Page 121).

2.2.8 Finite Sequences

We need to consider that when we implement Lucid, users will usually only provide a *finite* sequence of data tokens. To unify this idea with the notion of *infinite histories* Lucid implementations usually define an **eod** token, which indicates the "end of data". Semantically, all values following this token in a particular stream have the denotation $\perp$. The token propagates through most operations and can be intercepted at the output of a Lucid implementation to terminate the program.

2.3 Scala

Scala is a modern, hybrid language, being both fully object oriented and higher-order functional. Functions are first class values, encoded as objects. Furthermore there are so called `case` classes which provide functionality similar to algebraic datatypes and pattern matching. Currying and partially applied functions are available and the inbuilt concept of *generics* is closer to proper parametric polymorphism than what can be found in most major object oriented languages.

Scala compiles to pure Java bytecode and can run on existing JVMs, which makes it highly portable. A further advantage is that Scala is fully compatible with existing Java libraries and can import them seamlessly. Finally, most compiled Scala class files can be imported into new Java projects.

All of these features make Scala a promising language in its own right but the reason for its use as the implementation language in this project is the existence of two core libraries. The first is the parser combinator library and the second is the native concurrency abstraction based on the asynchronous *Actor Model* [8].

Function	CFG symbol	Combinator
literal terminal	`'lit'`	`"..."`
regular expression terminal	n/a	`"...".r`
sequential composition	`P Q`	`P ~ Q`
sequential composition, keep left/right only	n/a	`P <~ Q`, `P ~> Q`
alternatives	`P | Q`	`P | Q`
option	`[P]`	`opt(P)`
repetition	`{P}`	`rep(P)`
interleaved repetition	n/a	`repsep(P, Q)`
result conversion with function `f`	n/a	`P ^^ f`

Table 2.1: Overview of Scala parser combinators, P and Q range over parsers

2.3.1 Parser Combinators

When using parser combinators, parsers are first class values. In Scala they are encoded as functions from input readers to parsing results. A result can be a success, optionally carrying some value that was extracted along the way, e.g. a calculated number or an AST, a failure or an error, both with respective diagnostic messages.

A library of combinators then provides a set of algebraic operators which take parsers as operands and produce new and more sophisticated parsers as results. These operators are designed to mirror the connectives found in the formalism of Context Free Grammars (CFG) plus a few more as shown in Table 2.1 (which will appear in some code samples later).

CFGs are common for defining syntax and the source code of an implementation with parser combinators often looks similar to the original definition (in some cases it can be achieved by only a simple textual substitution). The advantage of this congruence is that the implementing source code directly documents the language grammar. This simplifies maintenance and later extension. Another advantage of parser combinators over external tools like Yacc[5], Flex[6] or Bison[7] is that there is no need to learn another tool-specific mark-up language.

In Scala, a combinator-based parser is a backtracking, recursive descent system with lazy evaluation. As a consequence we do not need lookaheads and recursively defined parsers are possible (e.g. parenthesis matching). We still need to avoid leftrecursion though.

[5] `http://invisible-island.net/byacc/byacc.html`
[6] `http://flex.sourceforge.net/`
[7] `http://www.gnu.org/software/bison/`

A detailed discussion of this framework can be found in ([15], Chapter 31).

2.3.2 Actor Concurrency Model

The decision to use actors is based on the strong congruence between a demand-driven dataflow network and the actor concurrency model, where nodes send data tokens along channels.

Scala's actor library layers over Java's thread model and hides the threading primitives. The underlying philosophy is that while Java threads are sufficiently powerful and expressive they are difficult to program with and reason about, and hence rather error prone.

Java threads are based on a *share-everything-and-use-locks* concept. Actors on the other hand favour a *share-nothing* approach. Due to its backwards compatibility with Java, sharing cannot completely be eradicated by the language[8] but when one only uses the actor abstractions and adheres to a few best-practice rules it becomes much easier to reason about the correctness and thread safety of the program.

In this model actors represent concurrently operating entities. They have some private data structures and a so-called *mailbox*. Furthermore actors can search their mailbox for particular messages and then trigger the sequential execution of a message handler. Each actor also exports a method which allows other entities of the system to put a message into that actor's mailbox. When messages are sent to an actor the sending method ensures safe publishing, i.e. the message appears atomically in the receivers mailbox. Most such systems provide a FIFO ordering between any two actors, while messages from multiple actors can be arbitrarily interleaved.

Scala provides two implementations of this concurrency abstraction. The first allocates one thread per actor. Its `act()` is a thin wrapper around the thread's `run()` method. The behaviour is implemented using a loop that continuously calls `receive()` which takes a partial function to walk the mailbox, picking out the first appropriate message using pattern matching. This triggers the invocation of the associated handler. When the extraction fails, the thread blocks until the next message arrives.

Java is fairly limited in the number of supportable threads, which led to the second implementation in the actor library. Here actors are detached from

[8] See languages like Erlang [1] which have chosen a pure actor model.

threads and allocated onto a fixed size thread pool. To obtain this behaviour, a special looping construct is used together with the `react()` method. This method will always cause an exception which the subsystem intercepts. At that point a closure of the actor state is taken and the call stack can be dismantled to free the thread for another actor. A detailed discussion is given in [8].

With the latter approach a single thread could theoretically run a program of arbitrarily many actors, given that no actor is ever calling `receive()`. To increase performance on a system that "has several processor cores, the actors subsystem will use enough threads to utilize all cores when it can" (page 590, [15]). This property is key to simLucid's scalability to multi-cores.

Chapter 3

Implementation

In this section I will present the implementation and operation of the simLucid interpreter. From a high level point of view (Figure 3.1) the system consists of a *Front End* (FE) and a *Back End* (BE), including an integrated *runtime system* (RTS). The FE lexes and parses a well-formed Lucid program into an *Abstract Syntax Tree* (AST). This AST is then passed to the BE which transforms the tree into a parallel, demand-driven dataflow network. The major components of the implementation will now be discussed in detail.

3.1 The Abstract Syntax Tree

The AST is the interface between the FE and the BE. Its implementation captures the full grammar shown in Appendix A.

The design mainly builds on two core features in Scala, `case` classes and the `sealed` keyword, which support Scala's pattern matching facilities.

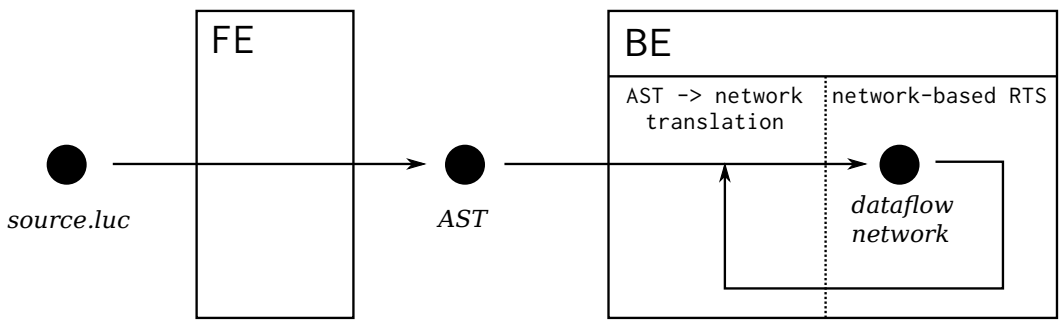


Figure 3.1: The high-level system layout.

ASTs do not contain any mutable state to enable safe passing between threads and simple deconstruction.

3.1.1 Constructing and Disassembling ASTs

I will use the encoding of binary operators to demonstrate the basic design of the data structure. The relevant type of tree node is defined like this:

```
...
sealed abstract class Expr() extends LuTerm
...
case class BinOp(op: String, l: Expr, r: Expr) extends Expr
...
```

The parent class `Expr` is marked `abstract` to prevent direct instantiation and the `sealed` keyword tells the compiler that all descendants of this class in the inheritance hierarchy can be found in the same file. This additional static information allows the compiler to perform exhaustiveness checks during a pattern match.

The inheritors of `Expr` are marked with the `case` keyword, which adds a few syntactic components implicitly to an ordinary class declaration. The only data a tree node should contain are the parameters passed in the constructor which should be immutable. The `case` keyword implicitly adds the syntactic constructs to achieve this. Most importantly though, the method `unapply` is added to the class which is used in a pattern match to get the actual parameters out of the object (i.e. a *deconstructor*).

Since Scala does all the groundwork the second line above is all we need to encode binary operators. Other components are treated similarly.

The following code demonstrates how nodes are deconstructed in the BE using pattern matching on the AST. Details will be given later but the crucial point here is that `e` is downcast to a `BinOp` and structurally disassembled

```
...
private def mapExpr(e: Expr,
                    envls: List[Environment],
                    log: mutable.Set[BaseNode]): BaseNode = {
  e match {
    ...
    case BinOp("+", l, r) =>
      val n = intAdd(log)
      n.leftInput = mapExpr(l,envls,log)
      n.rightInput = mapExpr(r,envls,log)
      n
    case BinOp("-", l, r) =>
      val n = intSub(log)
      n.leftInput = mapExpr(l,envls,log)
      n.rightInput = mapExpr(r,envls,log)
      n
    case ...
  }
}
...
```

3.2 The Front End

While classical parsers factor out two stages for lexical and syntactical analysis which are pipelined to process a stream of characters and later tokens, the layout of a combinator-based parser is quite different. The horizontal composition in classical systems is based on a separation of technologies, regular automata for lexing and context free grammars (CFG) for syntax analysis. This helps in dealing with ambiguity in the input language and reduces the size of parsing tables in the second stage. Combinator parsing libraries on the other hand merge the two technologies into one framework. The abstraction from characters to tokens is not lost but hidden in the implementation of the various combinators.

Consequently, combinator-based parsers tend to be vertically composed in an inheritance relation, where each class provides a collection of parser objects which are composed from smaller, more primitive parsers in the parent classes. I followed this approach and created two such classes,`Lexer` and `Parser`, which are related to the basic traits `Parsers` and `RegexParsers` as shown in Figure 3.2.

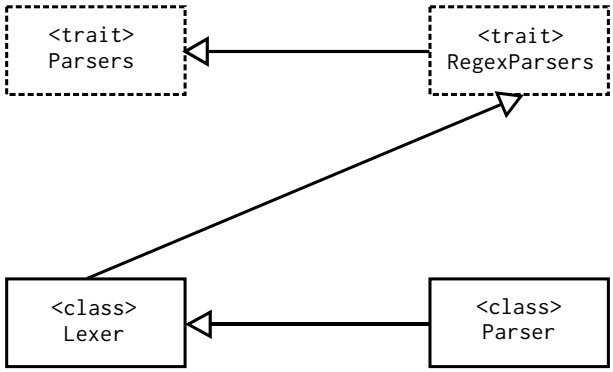


Figure 3.2: The classes Lexer and Parser extend the basic Parser hierarchy

The latter of the two traits provides regular expressions as parsers and handles skipping of whitespace in the input.

I distributed the various parsers over the two classes based on the following rule: the parsers for identifiers, integers, index and eod which only wrap around simple regular expressions and represent leaves in the AST have been placed in the `Lexer` class while everything else went into the `Parser` class. The split only serves readability purposes.

The core functionality sits in the `Parser` class and will be demonstrated with the following example fragment which deals with the binary, right associative, infix operator **`fby`**.

```
class Parser extends Lexer {

  def pred6_stream: Parser[Expr] = {
    (rep(pred5_stream~"fby")~pred5_stream) ^^ reduceRightAssocList
  }

  val reduceRightAssocList: List[Expr ~ String] ~ Expr => Expr = {
      case ps ~ i => (ps :\ i)(reduceRightAssoc)
  }

  def reduceRightAssoc(l: Expr ~ String, r: Expr) = BinOp(l._2, l._1, r)

}
```

For clarity I have removed debugging constructs.

At some point the FE will try to use the `pred6_stream` parser to consume some input. During the call the parser will be expanded to the body of the definition, which effectively asks for a sequence of `pred5_stream` non-terminals, separated by the literal "fby" (recall from Table 2.1 that ∼ means sequential composition). Note that an equivalent expression for the body could be `pred5_stream∼rep("fby"∼pred5_stream)`. The choice is based on the associativity of the operator. The encoding used for **`fby`** lends itself to right associativity while the alternative is better for left associative operators.

If this process succeeds, we have extracted a construct of type `List[Expr∼String]∼Expr`. However, we require a single `Expr` (or more concretely a `BinOp`) node. This is achieved through two helper functions which establish the correct associativity. The type `A∼B` is the infix notation for `∼[A,B]` which is operationally equivalent to pairs, providing standard projection methods `._1` and `._2`.

It is irrelevant whether we use `val` or `def` for defining function values.

The `reduceRightAssoc` function combines two `Expr`s and a string label into a single `BinOp` node. Meanwhile `reduceRightAssocList` performs a structural split of its argument to obtain the trailing `Expr` node and use that as the base case for right-folding `reduceRightAssoc` over the remaining list of (node, label) pairs. The `:\` operator is Scala's symbol for a right-fold.

For left associative operators, mirrored versions of these functions are used.

3.3 The Back End

The BE of the simLucid interpreter uses demand-driven dataflow networks to evaluate a Lucid program. The following section discusses the two main facets of this approach. Firstly, how are Lucid's lazy dataflow semantics realised on top of Scala's actor-based concurrency abstraction using demand-driven dataflow? And secondly, how can we systematically construct such networks from an AST?

3.3.1 Actor-based Dataflow Networks

Data-driven vs Demand-driven

There are two types of dataflow networks. Firstly, there are data-driven systems where inputs and source nodes are pushing values into the network and internal

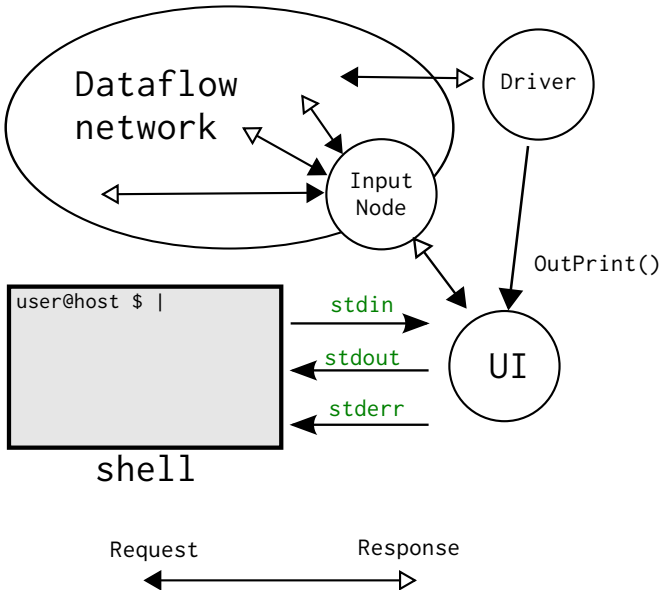


Figure 3.3: The overall layout of the BE at runtime

nodes perform their operation whenever a token is available on each incoming channel. Secondly, there are demand-driven networks where computations are initiated at the output, which send demands upstream and pull tokens back downstream. The latter approach corresponds to lazy dataflow semantics since demands are only generated for tokens which are actually needed. This approach is referred to as *eduction* in Lucid literature [19, 7], as is the approach taken here.

Overall layout

Figure 3.3 presents the overall layout of the dataflow networks constructed in the BE. There are two special nodes in the setup, the UI and the Driver[1] which together form the backbone of the RTS.

The UI node is the only entity in the whole system that interacts with the shell. This, obviously, is a bottleneck in the design, especially when the interpreter is run inside an OS pipeline. I tried a distributed approach first but discovered that Scala's read and write streams connected to the shell are not thread safe.

The Driver node generates a sequence of demands which are sent to the node inside the network that represents the top-level Lucid term. Each reply obtained

[1]Some publications would refer to this entity as the *educer*, but I prefer the term Driver since it "drives" the evaluation process.

is forwarded to the UI for printing. The demands are generated one at a time and when one demand has been answered the next one is dispatched.

A network may also contain zero or more input nodes (only one is shown in the figure), which correspond to free variables in the Lucid program. Any demands received at these nodes are redirected via the UI to the user or pipeline feeding the interpreter.

Not shown in the figure above is a `Debugger` object which is known to all components in the system and serves as a central switchboard for error handling.

Communication

Each node in the network is represented by an actor for which the Scala concurrency library provides methods for asynchronous sends and also a single primitive for a synchronous two-way communication. Since a synchronised handshake severely limits the opportunities for concurrent execution I almost exclusively used the asynchronous paradigm. The only exception to this is the modification of the network at runtime, where a limited degree of serialisation is unavoidable.

The basic, underlying communication paradigm used in the BE is a Request-Response scheme as shown in Figure 3.4. Based on Lucid's semantics, each node in the network corresponds to a whole (possibly multi-dimensional) sequence of values and the node receiving a `Request` has to know which of those values it is supposed to serve. For this purpose, the incoming `Request` carries a `Tag` (e.g. `tag_C` or `tag_D` in Figure 3.4) which can be understood as an index into the receiving node's data space. A tag contains two stacks of numbers, implemented with linked lists. The first is used for the time indexing. Most components will only look at the head. The only exception are *where clauses* containing declarations. For these a new timestamp is pushed onto the stack when the clause is entered and a timestamp is popped when a request leaves via a declared variable. The second stack is currently unused (see "Further Work" in Chapter 5).

Since communication is based on asynchronous (i.e. non-blocking) message sends, nodes require a way to match responses to requests. This is achieved with the second `Request` parameter. The `locId` is a unique number generated by the downstream node and enclosed in the request. The number is recorded by the receiving node and used to label the response when the desired data token becomes available. In the meantime, the downstream node sets up internal data structures specifying call-back hooks to deal with the response.

In Scala, any object can be sent as a message, but following best-practice

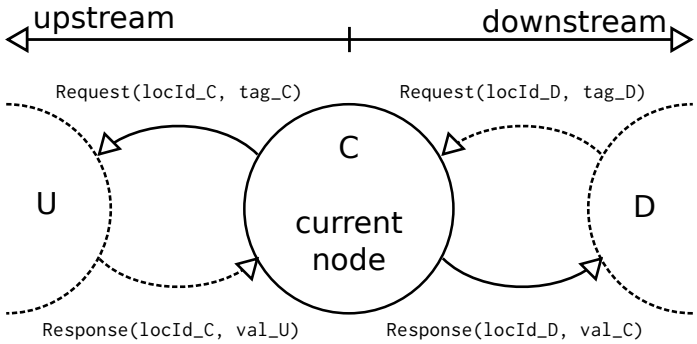


Figure 3.4: The basic communication pattern of BE nodes

design rules, I set up an `abstract sealed Message` class and derived a set of case classes from it[2], mainly to ensure thread safety. Apart from the two mentioned above (`Request`, `Response`) there is also a `Control` message carrying an integer control code (e.g. for node shutdown), the `OutPrint` message used by the `Driver` and two unused messages (`Diag`, `Status`) which were designed for later extension.

Basic Functionality

A lot of the functionality of the various types of network nodes is shared. To avoid code duplication this was implemented in the `abstract BaseNode` which extends Scala's `Actor` class and from which the more specific implementations inherit. A schematic representation is given in Figure 3.5.

A node starts executing in the `act()` method (bottom left) and first invokes `onStartUp`, followed by a construction of the runtime action stack. The stack is the partial function used to examine the mailbox. The main loop then repeatedly tries to extract and handle a message. Most nodes will need to perform some amount of work before being able to respond. To keep track of the progress of such a *job* an ID generator and a lookup table are provided as bookkeeping facilities. When a token can be served directly (e.g. cached, known at compile time) nodes bypass this job record mechanism. The four methods enclosed in dashed here lines are intended to be overwritten in concrete node classes. Some nodes also override the `finishJob` method.

The empty versions of these methods allow us to use the design pattern of *stackable modifications* ([15], Section 12.5). These modifications hook into the entry and exit communication pathways of a node and will later enable the modular implementation of caching and request aggregation.

[2]Similar to the AST data structure

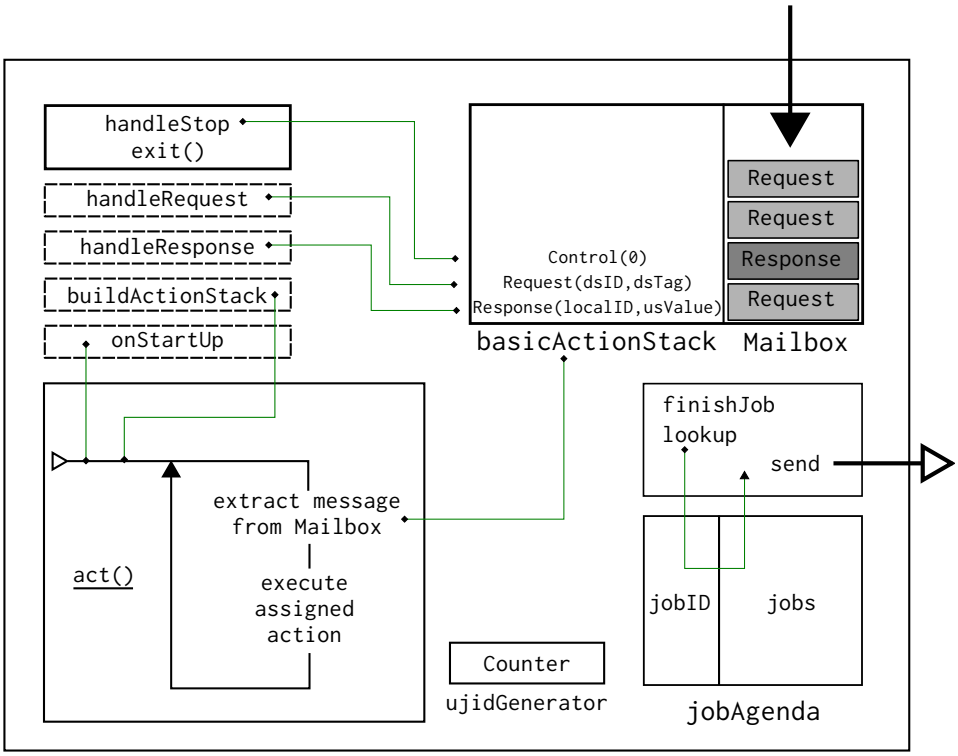


Figure 3.5: The BaseNode class on the inside

Figure 3.6 shows the upper section of the node inheritance graph and gives an overview of the various node types provided. Nodes are constructed in a factory, which can augment them with modifications and provide more specific subclasses where required (e.g. an adder derived from `IntegerArithmeticNode`).

The overall composition makes heavy use of Scala's traits which provide a limited degree of multiple inheritance. Traits encapsulate functionality, rather than representing an entity. When a trait is mixed into a class, its functionality is added to all objects derived from this composition. I moved the capability to request one, two or three inputs from upstream nodes into three separate traits. Apart from the ports itself those traits come with ID generators and lookup tables to implement the Request-Response matching system described above. Additionally, requests can be assigned to *slots* of a job. Each node can use these slots as it sees fit, e.g. to distinguish left and right operand inputs.

The supervisor trait enables a node to be in charge of a subnetwork (create, launch, stop, destruct). This is used by the *driver* which constructs the full network, starting from the top level node. The other type of node that makes use of this capability is the *function* node.

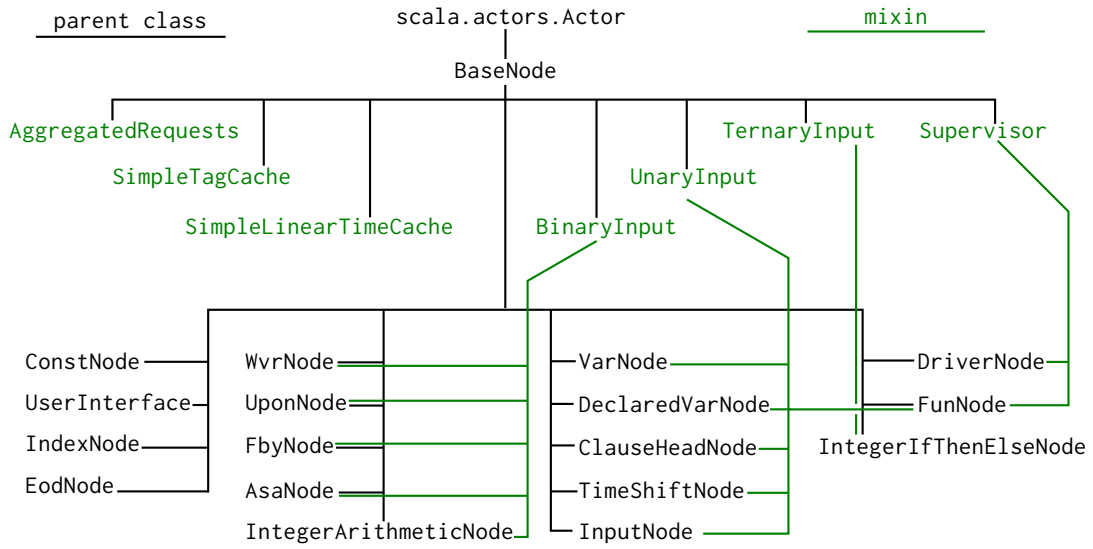


Figure 3.6: The Hierarchy of the various node types in the BE

3.3.2 Dataflow Nodes

Driver

At the time of creation the `Driver` node uses its capabilities as a supervisor to turn the full program AST provided into a network. Next the *driver* augements its action stack with an additional control method used in a simple shutdown protocol. Finally, the `onStartUp` method activates all the child nodes and then generates the first `Request` for time 0: `Tag([0],nil)`. This request is sent to the top-level node of the generated dataflow network.

When a response for time t is received, the node checks if the contained data is the special `TokEOD` token which indicates the end of the top-level data stream. If that is the case, the Driver initiates the shutdown protocol, which is just a ping-pong exchange with the UI to ensure that all outstanding `OutPrint` messages have drained. When the pong (`Control(10)`) comes back the Driver simply shoots the JVM.

Otherwise the data is wrapped into an `OutPrint` message, together with some header data and send to the UI. Meanwhile, a fresh tag for time $t+1$ is generated and a new demand is sent to the top-level node.

Constants

There are three types of nodes that serve values which are fixed at compile time.

The first is the `Index` node which serves the sequence $\langle 0, 1, 2, 3, 4, 5, \ldots \rangle$ and is implemented by just returning the demanded timestamp as an integer token.

Next there is the `Const` node which is given a data value at the time of construction and always responds with this value at runtime.

Finally, as a special form of the preceding node, there is the `EodNode` which always serves a `TokEOD`.

Extensional Operators

There are two base classes for extensional operators for the binary and the ternary case respectively: `IntegerArithmeticNode` and `IntegerIfThenElse`[3]. Unary extensional operators could be implemented similarly but were not required for the targeted subset of Lucid.

The *if-then-else-fi* construct is straight forward. It first asks the input guard stream for the current value and when the response comes back the data token is evaluated and a second request is send to the correct branch of the conditional. The result of that second request directly triggers the job completion routine and the data is piped straight through.

For all other extensional, binary operators the operation is handed down as a constructor parameter of the node (this specialisation is hidden in the factory) and stored inside each job. The jobs used for this purpose have two data slots and a query function that indicates whether a job is ready to compute a result.

Intensional Operators

At the moment the system provides native implementations for the two core intensional operators **`fby`** and **`next`** and also for the four most common derived ones: **`first`**, **`asa`**, **`upon`** and **`wvr`**. For all six the main task is to compute the correct timestamp to use on the data operand. After that, the resulting data value is piped straight through.

[3]The names are a slight misnomer since both classes are more general than the names suggest. They are there for legacy reasons only.

The `TimeShiftNode` implements **`next`** by simply incrementing the head of the timestamp stack by one, before forwarding the request to its input stream. **`first`** is implemented in the same way, but the shift function always sets the timestamp to 0.

The `FbyNode` checks if the timestamp in question is zero, in which case the left data operand will be asked to serve the value. Otherwise, the timestamp is decremented by one and the request goes to the right data operand.

The `AsaNode` comes with an internal loop that asks for guard tokens one at a time starting from $t = 0$ and keeps track of the current timestamp. As soon as the guard becomes true for the first time, the required timestamp is known and the data input can be queried.

The `WvrNode` and `UponNode` have similar internal loops and a persistent data structure to remember information obtained on previous requests. The first queries the guard until n true tokens have been seen (where n is the timestamp that was initially demanded) and keeps track of how far it got to provide a data time stamp. The second counts how many true tokens have been seen between 0 and $n - 1$ and uses that values as the data timestamp.

The operators **`wvr`**, **`upon`** and **`asa`** were implemented at a very late point in the design cycle. Slotting them in was easy and demonstrates the advantage of a well structured, modular design.

Definitions

`VarNode`s implement variable definitions and simply forward incoming requests unmodified to the top-level node representing the body of the definition. Results are equally piped through in the other direction.

Free Variables / User Inputs

`InputNodes`s are very similar to `VarNodes`. The only difference is that they do not have a core network node as their upstream neighbour but instead the UI. Since free variables are supposed to be globally unique, all such nodes are managed by the UI and allocated at demand, using synchronous communication between the constructing entity and the UI.

Furthermore, input streams are one dimensional thus input nodes use the top of the timestamp stack as an index, rather than the whole stack.

User Defined Functions

There are two ways to implement user defined functions. The simple approach constructs function applications as nodes with a single loose input. The node encapsulates a fragment of an AST and an environment. On the first demand a boolean flag ensures that the AST is converted into a section of a dataflow network, which is then tied to the loose input. When a function application node is created, the mapping between formal and actual parameters is encoded in the environment. This design is currently implemented but has one major drawback: recursive functions will generate a network that dynamically grows at runtime. This can quickly degrade performance.

The alternative (not implemented due to time constraints) is to assign a unique label to each *static* call site. The function body is then only mapped to a network once, but special nodes are required for each formal parameter. These formal parameter nodes have to be able to take in an arbitrary number of inputs and make routing decisions based on call site labels. The final requirement would be to ensure that downstream nodes are aware that their upstream peer is a function node and hence push the correct call site label into their tags.

Declarations

Declarations are implemented using the `DeclaredVarNode` which ensures transition from the inner to outer time dimension by popping one timestamp. At the other end, the top-level node of the subject of a *where clause* is prefixed with a `ClauseHeadNode` that simply duplicates the top of the timestamp stack and hence moves the evaluation into the inner time domain. These nodes are omitted if a clause does not contain any declarations.

3.3.3 Mapping from ASTs to Networks

The BE must translate the AST of a well-formed Lucid program into a network based on the components presented above. This is achieved through a `Factory` object which is in charge of the node construction and a `Mapper` object which enables the traversal of the tree.

Preliminary Setup

The top-level application extends the FE `Parser` class and tries to convert the source file into an AST. Assuming that this was successful, a runtime environment (`RTE`) is created. The `RTE` creates and launches a UI node, adjusts debugging and printing behaviour based on flags provided by the user, and then builds a `DriverNode` which triggers the first call into the `Mapper`. The subsequent activation of the driver will in turn activate the whole network[4].

The Factory Object

As we have seen in the previous section there is a wide variety of nodes, but most of them require initialisation. Furthermore, the initialisations are diverse, e.g. the arithmetic nodes need to obtain the function they are supposed to compute and Input nodes need to be told what input type they should be expecting. The latter involves minor parsing functions since users can only provide strings on `stdin`, and we need to extract integers, reals, booleans or chars.

Another thing that needs to be done is to plug in the stackable modification traits which equip the nodes with a limited degree of caching (see below) and more importantly the capability to aggregate multiple requests for a single tag behind a single job. This avoids redundant computations for requests which arrive between the start and completion times of an identical job.

The `Factory` hides all this behind a unified interface. The construction methods exported expect only a minimal number of parameters. It provides one further functionality: logging of newly constructed nodes. Whenever it is asked to produce a new node, it expects to be given a log (basically a mutable set of `BaseNode`s) to which the new node is added. The purpose of this log is to allow the supervisor that called into the `Mapper`, and thereby triggered a sequence of factory calls, to be aware of which nodes it is supposed to manage. A supervisor creates an empty log for this purpose and is guaranteed that all its children are recorded in the log when the construction call returns.

As an example consider the construction of an integer divider:

[4]The source code for the top-level application and the RTE class are given in Appendix B for reference.

```
object Factory {
  type CoreCache = SimpleTagCache

  def intDiv(log: mutable.Set[BaseNode]): IntegerArithmeticNode =
    logged[IntegerDivider](log) {
      new IntegerDivider()
    }

  private class IntegerDivider()
    extends IntegerArithmeticNode("idiv", (_ / _))
    with AggregatedRequests with CoreCache

  def logged[T <: BaseNode](log: mutable.Set[BaseNode])(con: => T):T = {
    val node = con
    log += node
    node
  }
}
```

The type declaration `CoreCache` provides a name alias to simplify the modification of the caching regime. The `intDiv` method is the construction routine which the factory exports to the outside world. It gives back an `IntegerArithmeticNode` which is an instance of the more specific internal `IntegerDivider` class. The second parameter given to the node constructor is an anonymous function object which takes two parameters and divides the first by the second.

The logger is implemented using a custom control abstraction which takes a type, a log and a value that has the given type. The last argument is defined as a *by-name* parameter and whatever expression is provided will only be evaluated when it is needed in the body of the abstraction.

The Mapper Object

The `Mapper` object provides a single method `mapToNode` which takes an AST, an environment and a log (presumably empty) and transforms the AST into a network of nodes. The top-level node of that network will eventually be returned.

The mapping is achieved by a collection of private methods that follow a similar pattern but only accept particular syntactic constructs (*Lucid terms*, *where clauses* and *expressions*).

Subcases are separated using pattern matching, e.g. LuTerms can only either be a WhereClause or an Expr (recall the use of sealed parent classes) and the appropriate mapping function is invoked. The collection of mapping functions builds a pre- and post-fixing tree walk framework that recursively processes an AST.

The two interesting methods are the one dealing with *expressions* and the one dealing with *where clauses*.

For *expressions* the method immediately enters a case split based on its argument, e.g.

```
case BinOp("div", l, r) =>
  val n = intDiv(log)
  n.leftInput = mapExpr(l,envls,log)
  n.rightInput = mapExpr(r,envls,log)
  n
```

Here we are matching a BinOp node with a label for integer division and two subexpressions. First, a divider node is built using a Factory construction method. Next we recursively convert the left and right subtrees into nodes and wire these to the inputs of our divider node, which is then returned.

Slightly more involved is the treatment of variables which have to be looked up in the current environment. The environment is actually a stack of Environment objects for each nesting depth. A helper method processes this list from head to tail and tries to resolve the identifier in each one of them until it finds a match. This implements the functionality that inner variable definitions shadow outer ones. When the list has been consumed without producing a match, the identifier is a *global* of the program and has to be mapped to a user input. The UI is in charge of all input nodes so a synchronous request is sent to the UI to procure the corresponding reference. The UI can match the request to an existing input node or creates a new one.

For function applications a similar lookup is required. Here environments store closures, rather than nodes. Function definitions are not global, so an error will be thrown if the lookup fails. Otherwise the obtained closure will be used as follows: a first check ensures that the number of actual parameters matches the number of formal parameters. Next the FunApp node, containing a list of ASTs which represent the actual parameters, is converted into a list of BaseNodes by mapping the function mapLuTerm(_,envls,log) over it. The result

is zipped together with the list of formal parameters and injected into a fresh environment object, which in turn is prepended to the environment found in the closure. At this point we can finally construct a function node using the augmented environment and the AST of the function body.

Finally, let us look at the `mapWhereClause` routine, which has to deal with the possible existence of declarations, mutual recursion between definitions and the creation of correct environments and function closures before the *subject* of the clause (an `Expr`) can be processed. Since Lucid specifies that the ordering of declarations and that of definitions are irrelevant (as long as all declarations precede all definitions) the translation is necessarily a three pass process.

On the first pass all identifiers are recorded in a new environment object. Declarations are immediately completed since their bodies use the outer environment, not the one being constructed, so we do not have to worry about recursion. For variable definitions, only a `VarNode` with an open input is created, while the body remains untouched. This is done, so that when the bodies are processed later, all variable nodes exist, even before their implementation is finished. In this fashion, loops can be introduced into the dataflow graph which correspond to recursive definitions. For functions, a partial closure containing the body AST and the list of formal parameters but not the full environment is added.

At this point we have a mapping from new identifiers to nodes or closures and can construct a new `Environment` object. The second pass then injects this new object into the full environment in each function closure and on a third pass all bodies of variable definitions are turned into networks using the augmented environment.

Lastly we use the new environment to translate the subject of the clause and, if the list of declarations was non-empty, also prefix it with a clause head node.

3.4 Extension: Caching

I have implemented a very simple caching strategy, where a cache is a mutable map from tags to data tokens. This map is located in a trait that can be mixed into any `BaseNode` and chains into the `handleRequest` and `finishJob` communication sequences (as does the `AggregatedRequests` trait). Caches use the `abstract override` keyword sequence together with `super` calls. Recall the factory example for the divider node:

```
private class IntegerDivider()
  extends IntegerArithmeticNode("idiv", (_ / _))
  with AggregatedRequests with CoreCache
```

The sequence of mixins determines the order in which `super` calls are resolved. Say a particular requests sits at the head of the mailbox of a divider node. The action stack in the `BaseNode` will invoke the `handleRequest` message. The most recent version of that method is the one defined in the cache trait, which on a success will immediately generate a response before the core of the node is even aware of this request. Otherwise, the `super` version will be called which is found in the request aggregation. If there is no ongoing job for this tag the next `super` call will eventually reach the version specified in the `IntegerArithmeticNode` and proceed as normal.

Once the node decides to finish the job the same resolution order occurs. The cache gets to go first and injects the return value into the map, then the aggregation component can inform all secondary requests about the outcome and finally the `BaseNode` (`IntegerArithmeticNode` does not override the `finsihJob` method) responds to the primary request and then disposes the of job.

Chapter 4

Evaluation

This section considers a variety of properties of the simLucid interpreter, both in absolute terms as well as in comparison to the pLucid system. The programs which where chosen to test and evaluate the interpreter are a selection from those delivered with the pLucid interpreter (with the exception of the hamming program which is only outlined in [19]). Among them they exercise all major language features supported by my implementation and produce outputs which are mathematically well-defined and hence can be verified automatically.

I will in turn look at the expressiveness, the correctness of the implementation and the performance of my interpreter.

4.1 Expressiveness

Table 4.1 compares the expressiveness of simLucid with the pLucid interpreter. The major difference is the selection of datatypes and associated operators which are available in the systems. For full details, refer to Appendix A.

The predicates to test for the type of a data token at runtime are left out since a static type checker was planned (see "Further Work" in Chapter 5). pLucid is dynamically typed and hence requires this mechanism.

Booleans are simulated in the system using the classical C convention that `0` represents `false` and any non-zero value indicates `true`. The associated operators can be implemented using function definitions and the *if-then-else-fi* construct.

Feature	pLucid	simLucid	Remarks
Integers	✓	✓	
Arithmetic operators	✓	✓	excluding exponentiation (`**`)
Relational operators	✓	✓	
Booleans + operators	✓	✗	Simulated using C-convention
Reals + operators	✓	✗	
Chars + operators	✓	✗	
Lists + operators	✓	✗	
Intensional operators	✓	✓	excluding `attime`
`eod` test predicate	✓	✗	
Type test prediactes	✓	✗	intentionally left out
`is current` Declarations	✓	✓	
User defined rec. functions	✓	✓	
`case` and `cond`	✓	✗	simulate using nested *if-then-else-fi*
`index` predefined	✓	✓	

Table 4.1: Summary of the expressiveness of simLucid in comparison to pLucid

Program	pLucid	simLucid	Remarks
`runavg`	✓	✓	
`runtotfun`	✓	✓	
`howfar`	✓	✓	
`factorial`	✓	✓	
`hamming`	✗	✓	pLucid fails at $ham(842)$
`fibonacci`	✗	✓	pLucid fails at $fib(36)$ and $fib(40)$
`primes`	✓	✓	only verified for first 1k primes

Table 4.2: Summary of the results of the correctness tests. Where both results are marked ✓, this indicates relative agreement, otherwise it signifies externally verified correctness.

4.2 Correctness

The results presented in this section are summarized in Table 4.2.

4.2.1 Running Average

```
next (s div n)
  where
    s = 0 fby s + x;
    n = 0 fby n + 1;
  end
```

The first test program computes the running average of a stream of numbers. The algorithm used is straightforward. We consume one element at a time and maintain a running total. Furthermore we count the number of inputs we have seen and the result for each consumed element is the current total divided by the current count. First tests were done by hand for small inputs and verified manually, while larger test sets of randomly generated input where feed into both interpreters which in all cases agreed on the output sequence.

This demonstrates that the Lucid operators **fby** and **next** are implemented correctly, that simple *where clauses* are dealt with correctly with respect to name resolution and also that recursive variable definitions are mapped correctly. The following tests make use of these features as well and further strengthen this result.

4.2.2 Fibonacci

The Fibonacci sequence is defined as

$$\begin{aligned} \text{fib } 0 &= 1 \\ \text{fib } 1 &= 1 \\ \text{fib } n &= \text{fib}(n-2) + \text{fib}(n-1) \end{aligned}$$

It can be shown by simple mathematical induction on the timestamps of demands that the definition of **F** in the following program generates exactly this sequence.

```
if index > first X then eod else
  F
    where
      F = 1 fby (F fby F + next F);
    end
fi
```

Since the sequence is self contained and could go on forever the test program wraps it into a conditional to terminate the output after N values have been produced where N is the first value provided by the user on the input channel **x**. On both interpreters the system terminated at the right time which demonstrates the correct implementation of conditionals and the **eod** token.

The two interpreters did not however agree on the output everywhere. Since the sequence is well defined mathematically, at least one of the outcomes had to be wrong. In particular the results began to diverge at $fib(36)$ where simLucid claimed $fib(36) = 24157817$ while pLucid said $fib(36) = 24157816$. A quick check using a Haskell implementation of the function as well as an examination of the values preceding the error revealed that pLucid was incorrect and simLucid was correct. In pLucid

$$fib(34)_{pl} + fib(35)_{pl} = 9227465 + 14930352 = 24157817 \neq 24157816 = fib(36)_{pl}$$

The error accumulated until jumping inexplicably by another 8 at $fib(40)$. It is not clear exactly what is going wrong in pLucid but it may be related to the internal representation of integers.

4.2.3 Howfar

```
howfar
  where
    howfar = if X eq 0 then 0 else 1 + next howfar fi;
  end
```

The howfar program demonstrates the correct implementation of *future references* using **next** and laziness. As mentioned earlier, defining a sequence in terms of its own future can make sense as long as the recursive definition is guarded, either by an outer **fby** or, as in this example, by a conditional. This program computes a countdown to the next zero on the input stream. Hence the output is known when the input is zero (namely zero as well) and preceding outputs can refer to this value.

As an example, consider

$$X = \langle 5, 1, 6, 0, 8, 3, 0, 1, 0, 7, 4, \ldots \rangle$$
$$\text{howfar} = \langle 3, 2, 1, 0, 2, 1, 0, 1, 0, h_n, h_m \ldots \rangle$$

If `X` is known only to the point shown, then all we know from an operational point of view is that $h_n = h_m + 1$. This knowledge is encoded in the design of the system and will be unrolled whenever the next zero is encountered, at which point a batch of output values will complete.

As above the output from simLucid for small data sets was verified using predefined pairs of input and output sequences, followed by a successful agreement check between simLucid and pLucid for larger sets of random data.

4.2.4 Running Total

```
tot(input)
  where
    tot(X) = S
      where
        S = 0 fby S + X;
      end;
  end
```

This program uses the same algorithm as that for the running average, but does not bother to count the number of input items and omits the division. The only interesting thing here is that the algorithm is wrapped inside a user defined function. The correctness was verified as above and demonstrates that simple user defined functions work as desired.

4.2.5 Factorials

```
F asa index >= X
  where
    X is current x;
    I = 1 fby I+1;
    F = 1 fby F*I;
  end
```

This program computes the pointwise factorial of each number of the input stream as follows. `X is current x` freezes the input stream at each outer step while a counter is incremented until it reaches `X` and multiplied onto a running total in the nested time domain. The `asa` in the subject of the clause eventually extracts the value and sends it as the outer result to the output. Again a number of small examples where tested by hand and a successful agreement check on random data confirmed the correctness of the implementation of declarations.

4.2.6 Primes

```
if index < first x then prime else eod fi
  where
    prime = 2 fby (n whenever isprime(n));
    n = 3 fby n + 2;
    isprime(n) = mynot(divs) asa myor(divs, prime * prime > N)
      where
        N is current n;
        divs = N mod prime eq 0;
      end;
    mynot(b) = if b then 0 eq 1 else 1 eq 1 fi;
    myor(a,b) = if a then 1 eq 1 else b fi;
  end
```

The prime program was tested and verified similarly to preceding tests but only to a limit of 1000 output values (rather than the usual 10k) since the algorithm is not very efficient. It tests all odd numbers $N > 3$ for prime divisors up to $\sqrt{N}$. The method becomes fairly expensive, especially since my treatment of functions and declarations is not optimised in any way.

For this reason the program was not used for performance testing later. It did, however, reveal a hidden bug in my implementation that required me to change the mapping of *where clauses* from a 2-pass to a 3-pass process. In the original version, functions could be used before their closures were complete, which led to undefined values in the associated environments.

4.2.7 Hamming

```
h
  where
    h = 1 fby merge(merge(2 * h, 3 * h), 5 * h);
    merge(x, y) = if xx <= yy then xx else yy fi
      where
        xx = x upon xx <= yy;
        yy = y upon yy <= xx;
      end;
  end
```

The hamming program (Chapter 2) fails in pLucid at $ham(842)$. pLucid returns $ham(842) = 17578124 = 2^2 4394531$: neither 2, 3 nor 5 divide the remainder, so the number is not 5-smooth. According to simLucid, however, $ham(842) = 17578125 = 3^2 5^9$, which is a valid 5-smooth number, so simLucid is correct. Again, I can only speculate about the source of this error.

4.3 Performance

To evaluate the performance of my system I set up the following procedure. A small C program is used (Source in Appendix C) to print a timestamp with microsecond resolution to `stdout`. This is done before and after each run of any test case and the difference is taken as a rough approximation of the command's execution time. Each of the programs mentioned above (excluding the one for primes) was run with data sets of various sizes and appropriate formatting on each of the two interpreters. Each particular experiment was repeated 5 times and the arithmetic mean and variance where computed to obtain a data point. Error bars in the following graphs represent one standard deviation.

4.3.1 Machine used for Testing

All timings where taken on a 2.4 GHz Intel Core 2 Quad workstation (64 bit architecture) with 8GB RAM, running Fedora 9 (kernel version 2.6.27) with Gnome 2.22.3. During the experiment the machine was offline and only default system processes were running in the background. The underlying JVM was OpenJDK's JVM version 1.6 and Scala was used at version 2.7.7.

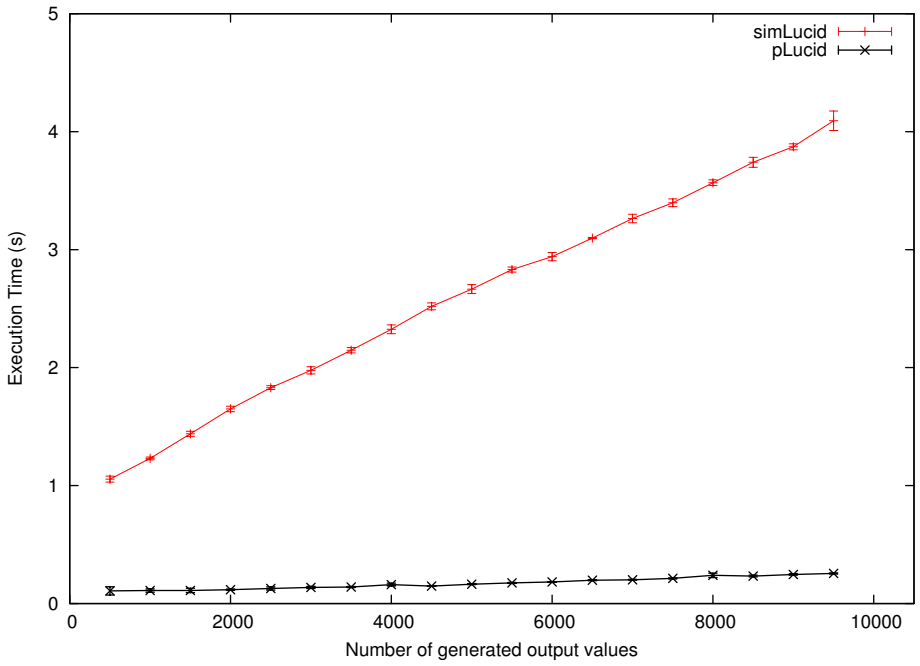


Figure 4.1: Performance results for the running total program

4.3.2 Absolute Runtime

All programs used for evaluation have shown a roughly linear increase in absolute execution time, with increasing gradients and startup times proportional to the complexity of the input program. In all cases, simLucid is beaten by pLucid in absolute terms. This comes as no surprise, since pLucid is highly optimised while simLucid is not yet optimised and runs above a JVM and several layers of library abstractions. Absolute performance, however, was not a primary goal of the implementation and hence this is not considered a drawback. Furthermore pLucid's incorrect output in the cases described above renders the fast computation speed irrelevant.

Note that the times recorded for simLucid come with about 0.1 to 0.2 seconds overhead on the test machine. This time is spent solely launching the JVM and loading necessary libraries as measured for an empty Scala application.

Figures 4.1, 4.2 and 4.3 show the results for `runtotfun`, `runavg` and `howfar` respectively. All of these programs do not exhibit a high degree of complexity (their dataflow networks are comparatively small) and their runtime behaviour is rather predictable.

The `hamming` program (Figure 4.4) is slightly more complex than `runtotfun`,

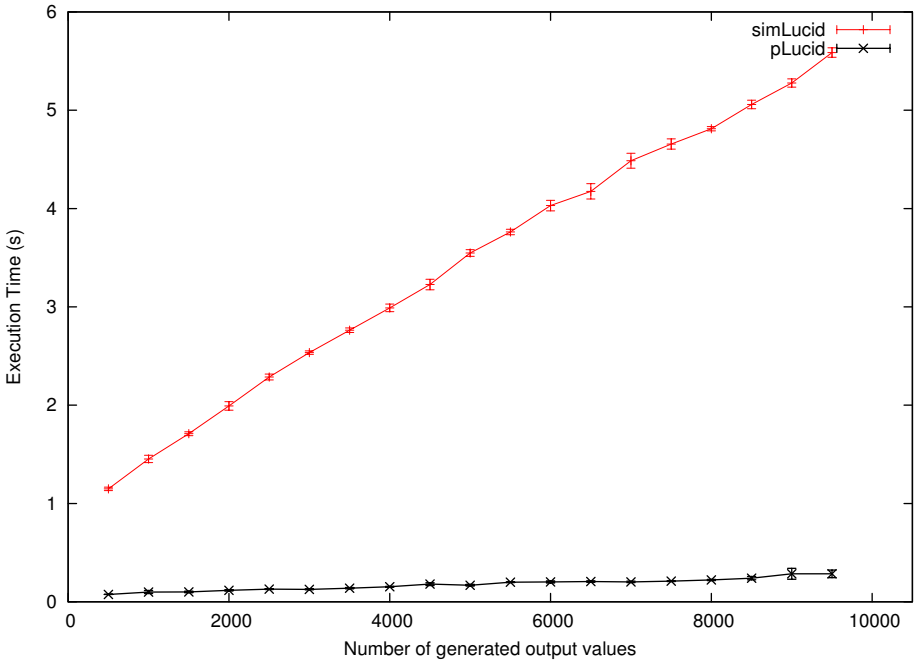


Figure 4.2: Performance results for the running average program

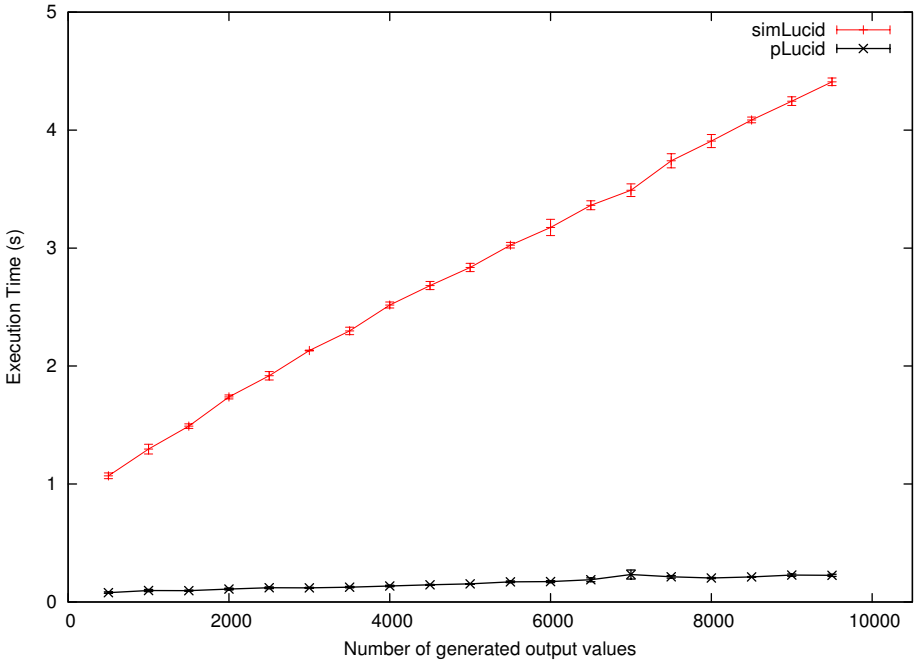


Figure 4.3: Performance results for the howfar program

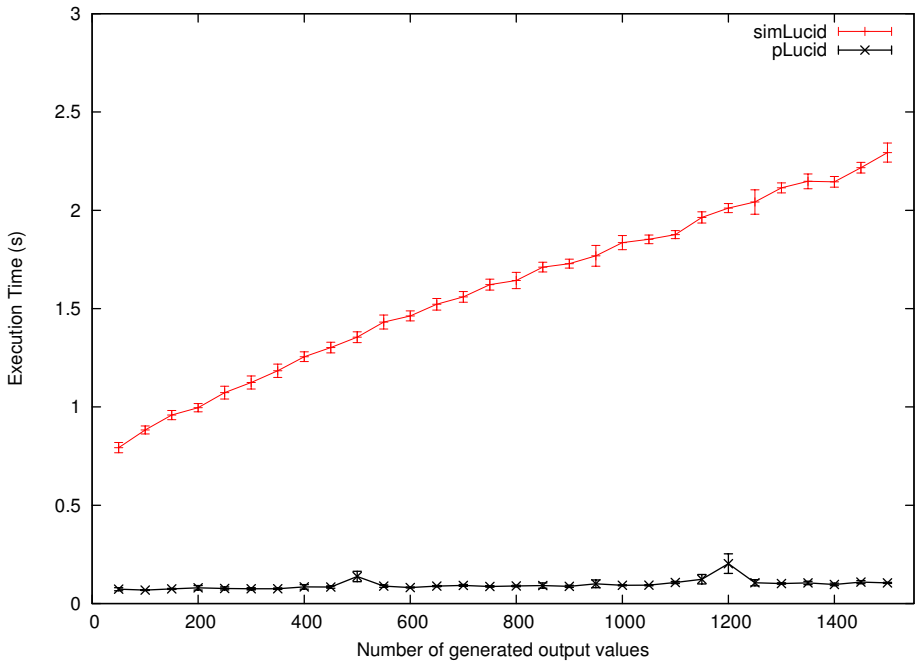


Figure 4.4: Performance results for the hamming program

runavg and **howfar** while actually running faster than the previous three. The key difference here is that in the three preceding examples each output value required the consumption of an input value. These input values were served via a UNIX pipe from a file on the local disk. **hamming** on the other hand only consumes a single value used as a cut-off point for the output generation. This highlights the problem of the UI bottleneck where both input and output have to pass through the same actor.

Similarly self-contained is the **fibonacci** program (Figure 4.5). Its network is much smaller, but the minimal increase in execution time for additional output elements has to be accredited to the implemented caching subsystem. This leads to an efficient runtime pattern that is comparable to typical array-based implementations in imperative languages (as opposed to the naive purely functional $O(2^N)$ implementation).

Finally, the **factorial** program exhibits noticeably longer runtimes (Figure 4.6), due to its two dimensional data space.

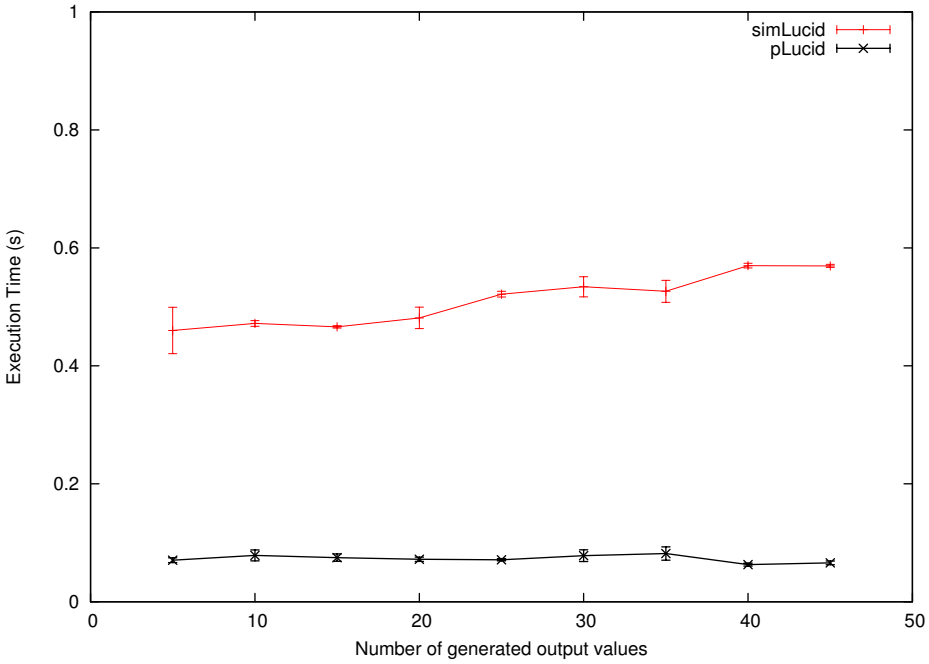


Figure 4.5: Performance results for the Fibonacci program

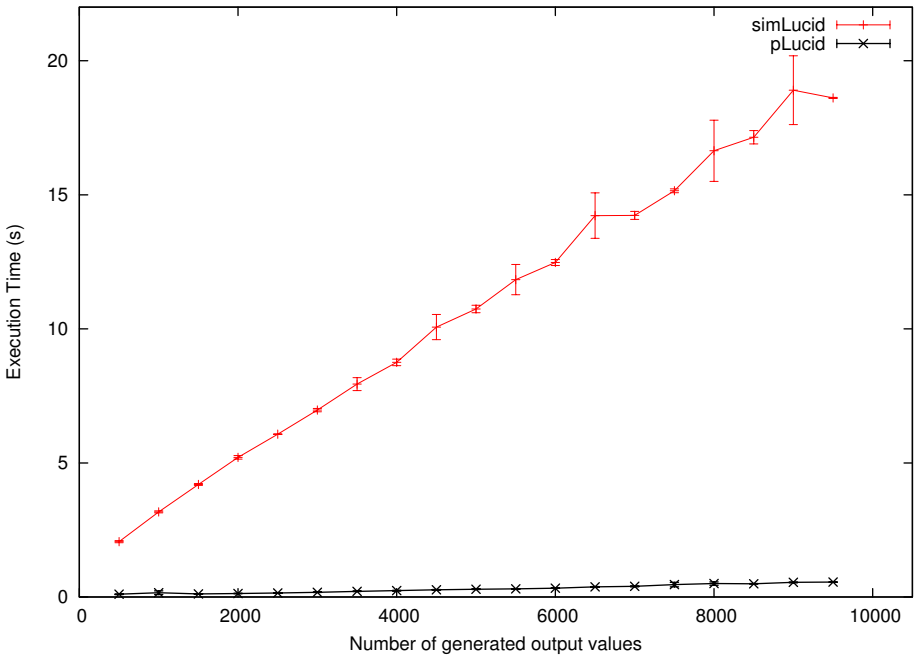


Figure 4.6: Performance results for the factorial program

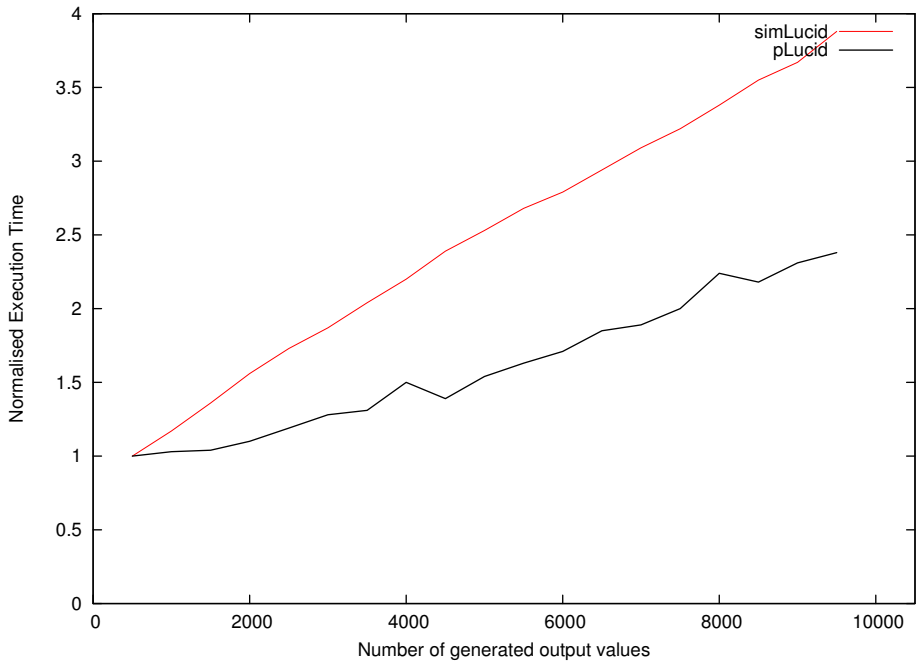


Figure 4.7: Normalised performance results for the running total program

4.3.3 Scaling Behaviour

To better compare the two systems, I normalised the running times of the tests with respect to their first data points. This revealed two interesting behaviours. Firstly, simLucid's runtime increases faster then that of pLucid. Secondly and more interestingly, simLucid scales up much more smoothly, while pLucid exhibits a fair amount of jitter. One has to take into account though that small changes introduced through varying OS activity have a much higher relative impact on the comparatively small times measured for pLucid. The results are shown in Figure 4.7 (**runtotfun**), Figure 4.8 (**runavg**), Figure 4.9 (**howfar**), Figure 4.10 (**hamming**), Figure 4.11 (**fibonacci**) and Figure 4.12 (**factorial**), which show the scaling behaviour of simLucid and pLucid with respect to input size.

Finally, I looked at the amount of parallelism inherent in the test programs and to what degree that could be exploited. I took three programs with varying degrees of complexity and executed experiments for each with the largest of the data sets used above. Using a Linux boot manager flag (`maxcpus=x`) I repeated this experiment with 1, 2, 3 and 4 active processor cores. The absolute and normalised times obtained are presented in Table 4.3. The normalised results are also visualised in Figure 4.15 (**runavg**), Figure 4.13 (**hamming**) and Figure 4.14 (**factorial**) and highlight the fundamental difference between the

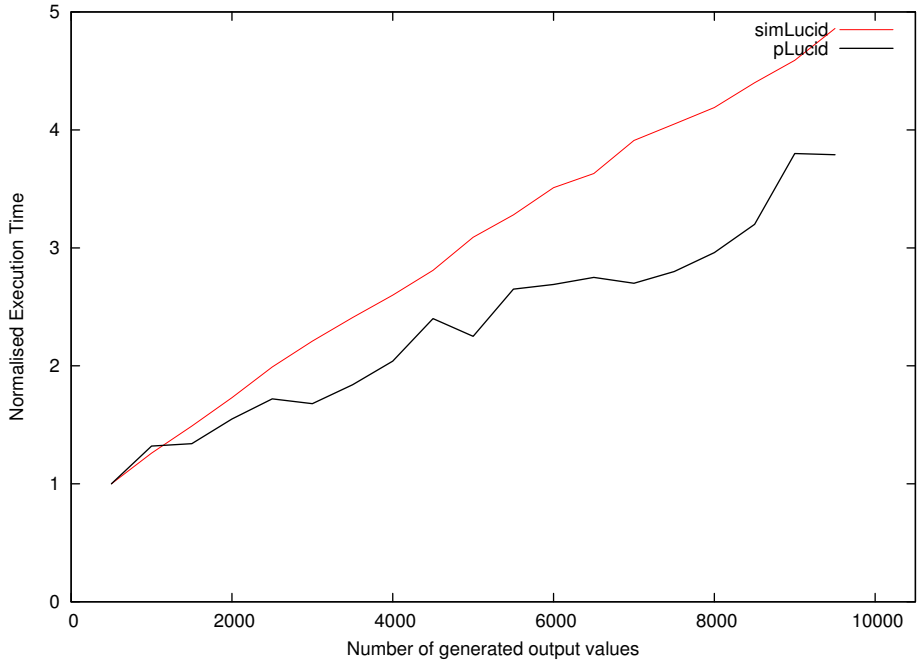


Figure 4.8: Normalised performance results for the running average program

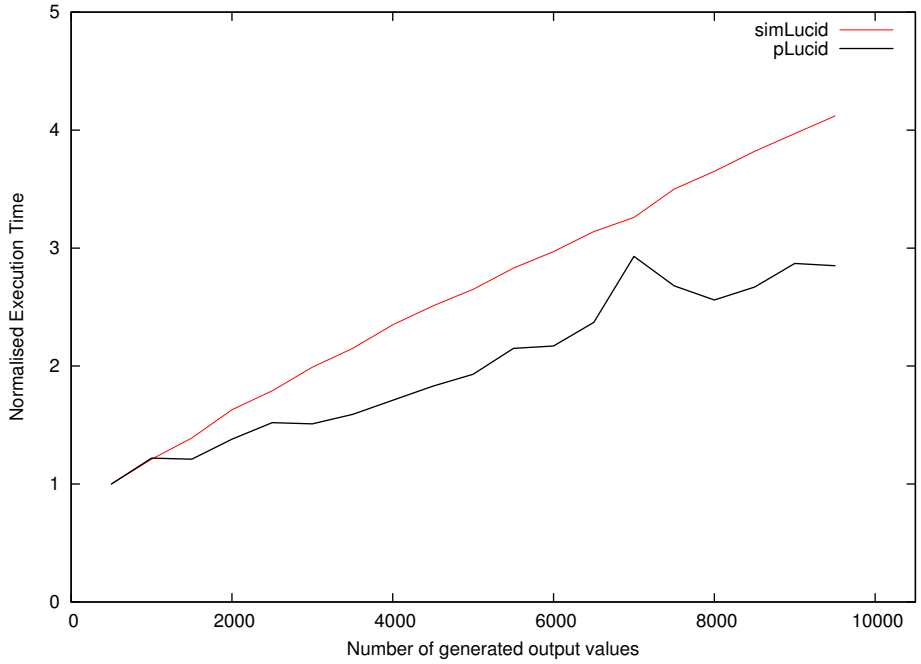


Figure 4.9: Normalised performance results for the howfar program

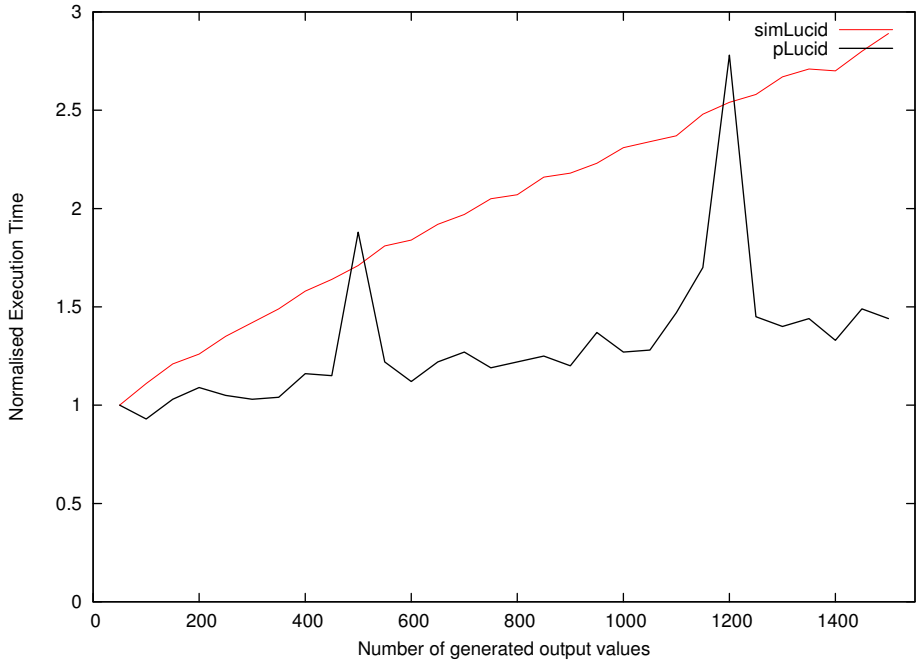


Figure 4.10: Normalised performance results for the hamming program

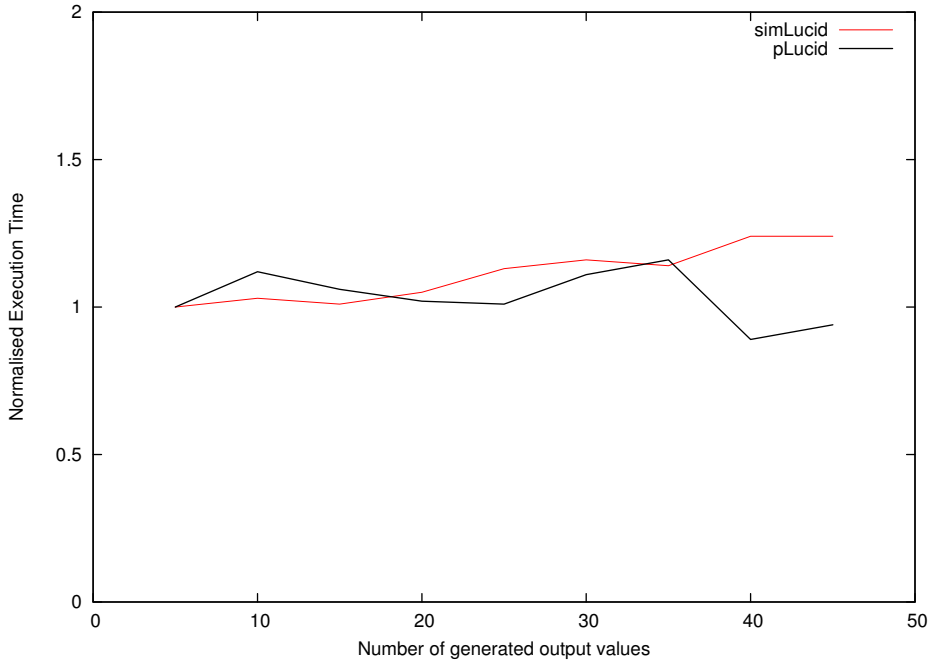


Figure 4.11: Normalised performance results for the Fibonacci program

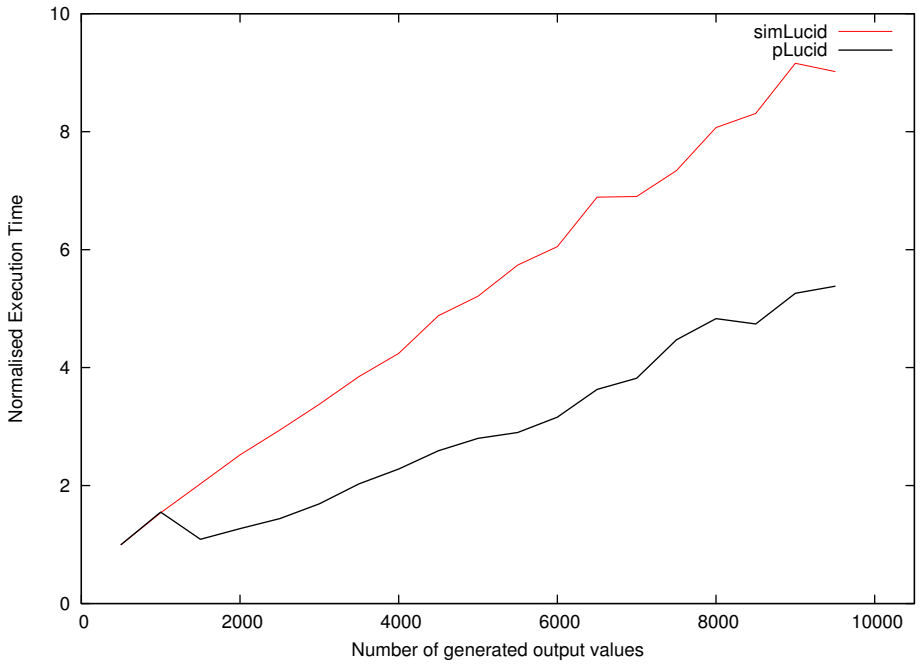


Figure 4.12: Normalised performance results for the factorial program

two interpreters.

pLucid is single threaded and cannot profit from any additional cores and hence its execution time remains unaffected by the number of cores (the 60% increase in the **hamming** case is again an effect of OS variations being overstated in the normalised result and only corresponds to a difference of about 70 milliseconds). simLucid on the other hand shows a marked improvement in all three cases when going from one to two cores. Here the actor subsystem increases the thread pool and hence enables the concurrent execution of multiple network nodes. In the case of **hamming**, which has the largest network among the three test cases, further improvements can be observed for three and four cores (about 10% from 2 to 3) while the other two do not gain any further speed-ups. Interestingly in all three cases, CPU utilisation dropped drastically when the speed-up curve began to flatten. An explanation for this is the limited amount of parallelism inherent in the examples used here. The **hamming** curve however looks rather promising, considering that the program is still fairly small.

	`hamming`		`factorial`		`runavg`	
Cores	pLucid	simLucid	pLucid	simLucid	pLucid	simLucid
absolute time / seconds						
1	0.12	6.08	0.60	31.80	0.25	9.08
2	0.13	3.17	0.60	21.36	0.29	5.91
3	0.14	2.56	0.58	21.15	0.26	6.14
4	0.20	2.43	0.58	19.87	0.31	5.66
normalized with respect to one core						
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.08	0.52	1.01	0.67	1.14	0.65
3	1.16	0.42	0.97	0.67	1.05	0.68
4	1.62	0.40	0.97	0.63	1.22	0.62

Table 4.3: Summary of the scaling behaviour of simLucid and pLucid with respect to the number of processor cores. For each program, the largest test data set was used.

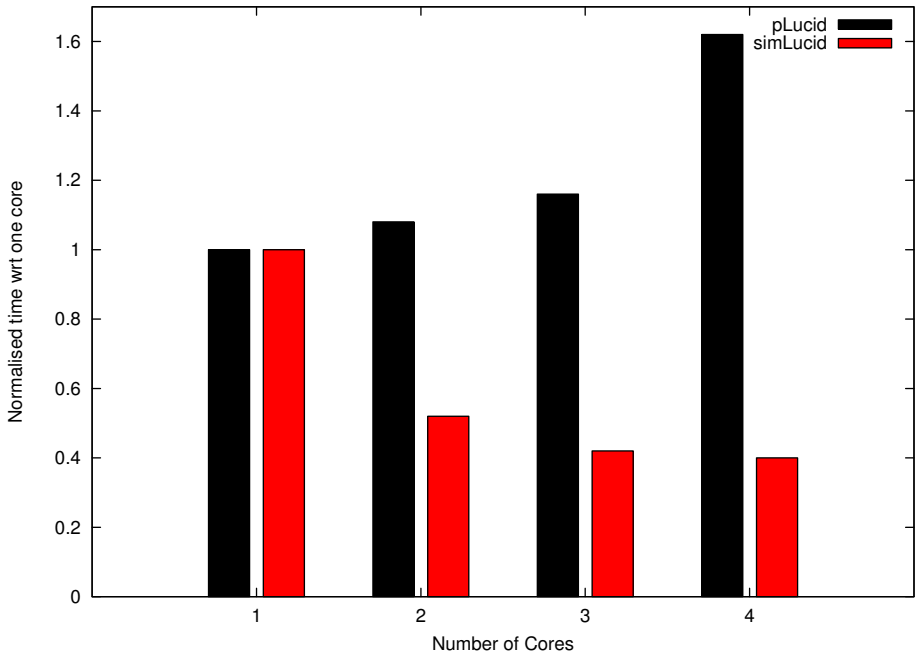


Figure 4.13: Scaling behaviour of the hamming program with respect to the number of cores

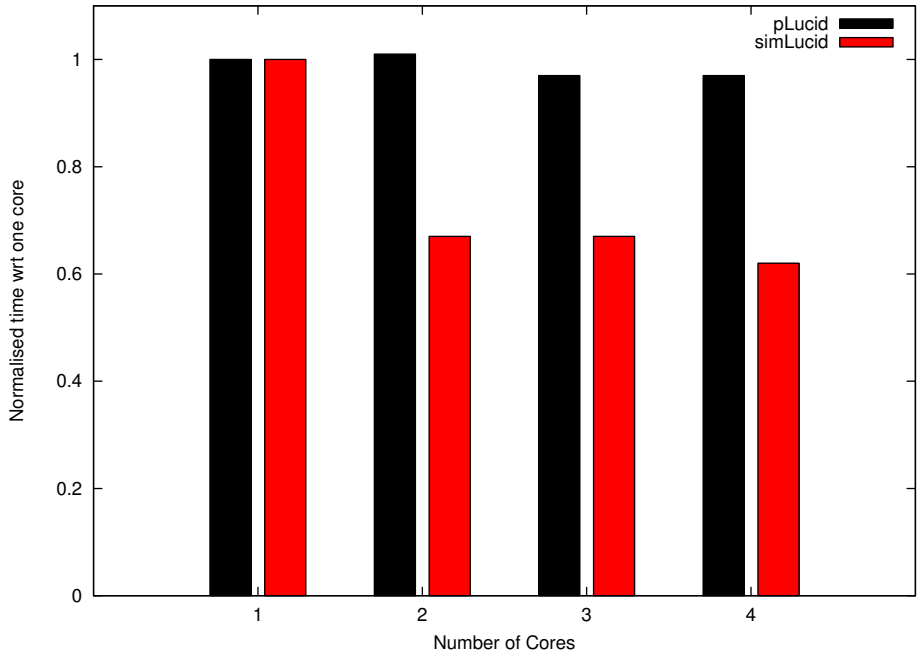


Figure 4.14: Scaling behaviour of the factorial program with respect to the number of cores

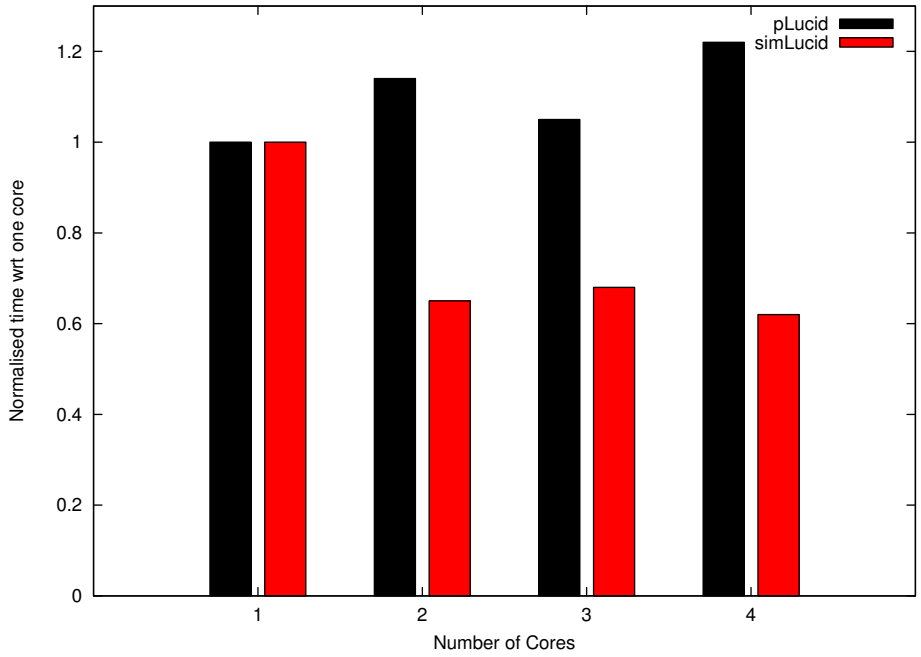


Figure 4.15: Scaling behaviour of the runavg program with respect to the number of cores

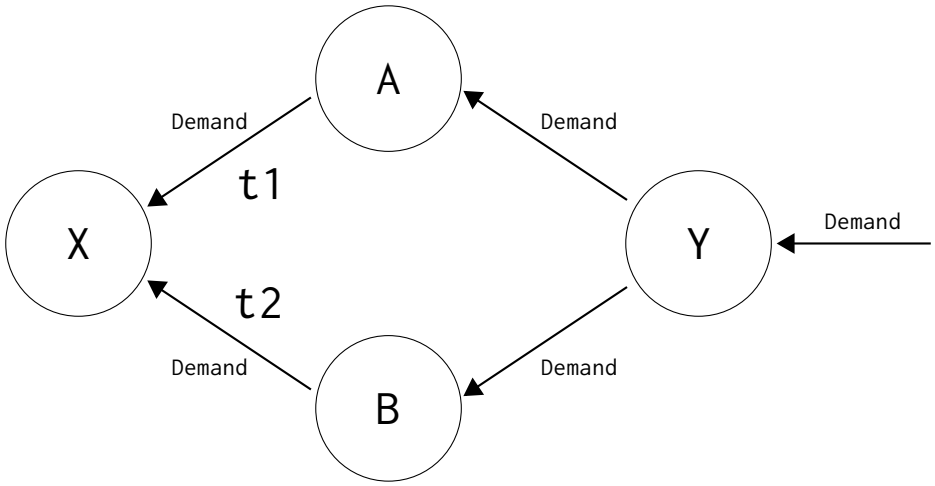


Figure 4.16: A diamond-shaped demand pattern

4.4 Non-deterministic Input Requests

Figure 4.16 demonstrates how the use of concurrently operating actors leads to non-determinism in the evaluation order. It is fairly common to have network structures where, for the general case, we *cannot* assume $t_1 = t_2$. Since Lucid is side-effect free this absence of a deterministic evaluation order is not a problem and allows us to exploit parallelism in the first place[1]. This assurance, however, breaks when the interpreter is interacting with the user. Inputting data is a side-effect.

Say node **X** is an input directly tied to the UI and a demand at **Y** generates two demands at **A** and **B** which in turn generate demands for **X** at times t_1 and t_2 respectively. Let us further assume that $t_1 \neq t_2$, and since **X** is an input both requests will eventually be presented to the user. Now since all four nodes are distinct, concurrently operating entities there is no guarantee about which of the two demands at **X** arrives and will be serviced first.

When running in standalone mode this is not a major issue since the user is presented with the explicit timestamps required, and from there on assigning the right input values to the right timeslots ensures correct evaluation behaviour. If, however, the interpreter takes its input from another UNIX process via a pipe, then this uncertainty in ordering will be observable at the output. Future work is to rectify this.

[1]Note that pLucid follows a strict left to right evaluation policy when presented with multiple evaluation options

Chapter 5

Conclusion

In my project I have successfully implemented a modern, high-level Lucid interpreter for the targeted subset of the language specification. My evaluation results demonstrate that all components are operating correctly with respect to Lucid's semantics as required by my success criteria. simLucid even supersedes pLucid in two cases in this regard.

Beyond that a number of extensions have been implemented, including multi dimensionality using `is current` declarations and simple caching. The modular design approach had major advantages for the development cycle, since it allowed quick modification of the supported language subset.

Furthermore an evaluation of the runtime performance of the simLucid system with respect to the pLucid interpreter was conducted.

5.1 Further Work

The interpreter is functioning correctly but there is still major scope for improvement. Further work could include the following.

The system was designed to be statically typed, and a small number of basic components have been implemented with this in mind. A typed AST would allow us to quickly add the missing data types to the system. Simultaneously, one could replace the basic message passing facilities with strongly typed channels [8] found in newer versions of the Scala actor library, possibly providing performance gains.

A second point to tackle is the UI bottleneck, which could significantly improve performance. This would either involve a thread safe solution for print and

write streams, or alternatively a clear distinction between who is allowed to read and who is allowed to write. When dealing with the UI one might also fix the non-determinism problem on the input by forcing the inputs to always request user data in sequential order. If they want to jump ahead they would be required to request the intermediate values as well and buffer them.

A performance boost could also be gained from a better caching system involving a garbage collector of some form and an effective dimensionality analysis to avoid duplicating working in multiple time dimensions in the presence of declarations. The latter is what crippled the performance of the `prime` program. Each prime was computed once for its own outer time and once for every other primes' outer timeslot.

A further area that could be improved is the implementation of functions. The approach would involve tagging each *static* occurrence of a function call with a callsite tag. Each formal parameter would then be represented with a variable capable of routing requests based on the callsite information associated with that request. This would allow fixed size networks at compile time, rather than dynamically unfolding ones. (This approach is taken in pLucid)

One could also widen the language specification to incorporate more advanced features like explicit multi-dimensionality [2], either in addition to or as a replacement of declarations. And finally, there is still the long term aim of turning the current BE into a full fledged many-core simulator.

5.2 Key Findings

The performance and scaling analysis showed that it is possible, to some degree, to abstract away the parallelism inherent in a program and leave the exploitation thereof to a distributed underlying runtime system. This project shows how well actors are suited for the implementation of parallel dataflow systems and gives credence to the idea of using Lucid for programming many-core machines.

Reflecting on the initial motivation, i.e. the long term transition to a hardware mapping, I believe it necessary to rigorously revise the Lucid language specification. In particular, user defined recursive functions and, more seriously, declarations that incur nested time seem to be ill-suited for a mapping to hardware. Even in a simulation domain it might be advisable to remove declarations since they drastically complicate the programming effort and dataflow implementation.

Bibliography

[1] J. Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[2] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multi-dimensional Programming*. Oxford University Press, Oxford, UK, 1995.

[3] E. A. Ashcroft and W. W. Wadge. The Syntax and Semantics of Lucid. Technical Report CSL-146, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, 1984.

[4] A. M. DeFlumere and S. R. Alam. Exploring multi-core limitations through comparison of contemporary systems. In *TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference*, pages 75–80, New York, NY, USA, 2009. ACM.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1976.

[6] A. A. Faustini, S. G. Matthews, and A. A. Yaghi. The pLucid Programming Manual. Technical Report TR84-004, Computer Science Department, Arizona State University, Tempe, Arizona 85287, USA, 1984.

[7] A. A. Faustini and W. W. Wadge. An eductive interpreter for the language Lucid. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 86–91, New York, NY, USA, 1987. ACM.

[8] P. Haller and M. Odersky. Actors that unify threads and events. In *COORDINATION'07: Proceedings of the 9th international conference on Coordination models and languages*, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41:33–38, 2008.

[10] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency: Practice and Experience*, 9(1):63–83, 1997.

[11] G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. In *IFIP Congress*, pages 993–998, 1977.

[12] P. J. Landin. The next 700 Programming Languages. *Communications of the ACM*, 9(3), MAR 1966.

[13] E. A. Lee. The Problem with Threads. *Computer*, 39:33–42, 2006.

[14] S. K. Moore. Multicore Is Bad News For Supercomputers, 2008. http://spectrum.ieee.org/computing/hardware/multicore-is-bad-news-for-supercomputers.

[15] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[16] J. Plaice, B. Mancilla, and G. Ditu. From Lucid to Translucid: Iteration, Dataflow, Intensional and Cartesian Programming. *Mathematics in Computer Science*, 2:37–61, 2008. http://www.springerlink.com/content/y5352j34q124r098.

[17] P. Rondogiannis and W. W. Wadge. Intensional Programming Languages, 1998.

[18] J. Stokes. Analysis: more than 16 cores may well be pointless, 2008. http://arstechnica.com/hardware/news/2008/12/analysis-more-than-16-cores-may-well-be-pointless.ars.

[19] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

Appendix A

The simLucid Grammar

The simLucid interpreter implements the following context free grammar. It makes use of the syntax for optional and/or repeated occurrances(square brakets and curly braces). The implementation is split over the two files *Lexer.scala* and *Parser.scala*.

program → lucid_term

lucid_term → expression [`'where'` { definition } { declaration } `'end'`]

definition → ident `'('` ident { `','` ident } `') ='` lucid_term `';'`
| ident `'='` lucid_term `';'`

declaration → ident `'is current'` lucid_term `';'`

expression → pred6_stream { `'asa'` pred6_stream
| `'upon'` pred6_stream
| `'whenever'` pred6_stream
| `'wvr'` pred6_stream }

pred6_stream → { pred5_stream `'fby'` } pred5_stream

pred5_stream → `'if'` lucid_term `'then'` lucid_term `'else'` lucid_term `'fi'`
| pred4_stream

pred4_stream → pred3_stream { `'eq'` pred3_stream | `'ne'` pred3_stream
| `'<='` pred3_stream | `'<'` pred3_stream
| `'>='` pred3_stream | `'>'` pred3_stream }

pred3_stream → pred2_stream { `'+'` pred2_stream | `'-'` pred2_stream }
pred2_stream → pred1_stream { `'*'` pred1_stream | `'mod'` pred1_stream
| `'div'` pred1_stream | `'\'` pred1_stream }
pred1_stream → `'next'` pred1_stream | `'first'` pred1_stream | pred0_stream
pred0_stream → fun_app | atom | `'('` lucid_term `')'`
fun_app → ident `'('` lucid_term { `','` lucid_term } `')'`
atom → index | eod | ident | integer
index → `'index'`
eod → `'eod'`

Appendix B

simLucid Source

B.1 File: SimLucid.scala

```
package sl.app.first

import collection._
import java.io.FileReader
import fe.first._
import be.first.RTE
import ast.first._
import util.first._

object SimLucid extends Parser with TypeChecker {

  private var sourceFile: String = "";
  private val flags = mutable.Map(
            "verbose"     -> false,
            "debug"       -> false,
            "no-eval"     -> false,
            "noInputFile" -> true)
  private val progName = "simLucid(1), version 1.0"

  def processFlag(f: String) {
    f match {
      case "--verbose" => flags += ("verbose" -> true)
      case "--debug" => flags += ("debug" -> true)
      case "--no-eval" => flags += ("no-eval" -> true)
```

```
    }
  }

  def main(args: Array[String]) {
    for(arg <- args) {
      arg match {
        case s: String if s.startsWith("--") => processFlag(s)
        case _ =>
          sourceFile = arg
          flags += ("noInputFile" -> false)
      }
    }

    Debugger.debugOn = flags("debug")
    Debugger.verboseOn = flags("verbose")

    import Debugger._

    if(verboseOn) Console.println(verbosePrefix+"running "+progName)
    if(verboseOn) Console.println(verbosePrefix+"flag settings: "+ flags)
    if(verboseOn) Console.println(verbosePrefix+"source file: "+ sourceFile)

    if(flags("noInputFile")) {
      if(panicOn)
        Console.println(panicPrefix+"no input found, system stopping ... ")
      exit
    }

    if(debugOn)
      Console.println(debugPrefix+"creating a reader on input file")
    val reader = new FileReader(sourceFile)

    if(debugOn) Console.println(debugPrefix+"parsing input")
    val parseResult = parseAll(program, reader)

    if(debugOn) Console.println(debugPrefix+"process parse result")
    val ast: Program = parseResult match {
      case Success(res, _) => res
      case Failure(msg, next) =>
        if(panicOn)
          Console.println(panicPrefix+
```

```
            "Parse failed: "+msg+"\n"+blockPrint(next))
          exit
        case Error(msg, next) =>
          if(panicOn)
            Console.println(panicPrefix+
              "Parsing error: "+msg+"\n"+blockPrint(next))
          exit
      }

      if(verboseOn)
        Console.println(verbosePrefix+
          "The parsed AST is:\n\n"+ast+"\n")

      if(debugOn)
        Console.println(debugPrefix+"Type checking AST -- not really atm")
      val typedAST = typecheck(ast)

      if(flags("no-eval")) {
        if(verboseOn)
          Console.println(verbosePrefix+
            "System running in no-eval mode, exiting now ...")
        exit
      }

      if(debugOn) Console.println(debugPrefix+"Setting up RTE")
      new RTE(typedAST)

    }

    def blockPrint(reader: Input): String = {
      val res = new StringBuilder(50)
      var remainder = reader
      while (! remainder.atEnd) {
        res.append(remainder.first)
        remainder = remainder.rest
      }
      res.toString
    }

  }
```

B.2 File: RTE.scala

```
package sl.be.first

import net._
import ast.first._
import util.first._

class RTE(val ast: Program) {

  val ui = new UserInterface()

  Debugger.verbosePort = ui
  Debugger.debugPort = ui
  Debugger.warningPort = ui
  Debugger.panicPort = ui

  Mapper.ui = ui

  ui.start()

  val driver = new DriverNode(ui,ast)

  driver.start()

}
```

Appendix C

Microsecond Timestamping

The following C program prints a timestamp with microsecond resolution to `stdout`. This relies on a Linux 2.6 kernel.

```
#include stdlib.h
#include stdio.h
#include sys/time.h
main()
{
  struct timeval tv;
  struct timezone tz;
  gettimeofday(&tv, &tz);
  printf("%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
  exit(0);
}
```

Appendix D

Project Proposal

Jonas Kaiser
Churchill College
jk431

Computer Science Part II Project Proposal

Reconsidering Lucid – a modern approach

October 22, 2009

Project Originator: Jonas Kaiser, after discussions with Alan Mycroft, Dominic Orchard and John Fawcett

Resources Required: See attached Project Resource Form

Project Supervisor: Dominic Orchard

Signature:

Director of Studies: John Fawcett

Signature:

Overseers: Graham Titmus and Alastair Beresford

Signatures:

Introduction and Description of the Work

Diminishing gains from the previous century's processor design methods have prompted designers to take a new approach: multicore machines. General purpose quadcore CPUs are now available as commodity hardware and recent developments in using manycore GPUs for general purpose computation highlight that programs must be targeted to run efficiently on manycore architectures.

If we run our existing programs on these machines, we will not see any performance increases (and indeed they may actually run slightly slower, as each core on its own might be weaker than a traditional single core CPU). Parallelising existing code or writing parallel code from scratch with the current mainstream programming tools on the other hand can unlock large portions of the available performance – if it is done correctly. But actually getting it right is far from easy with the tools at hand. The main reason here is that most mainstream languages were developed with the sequential von Neumann architecture in mind, i.e. we are using the wrong tools for the right task.

As the hardware is changing under our feet our mental model and the corresponding tools should change as well. It is not perfectly clear what this new model will be and hence it is necessary to investigate not just variations of the paradigm in use but to consider completely different paradigms as well. One alternative on which I want to focus is the *dataflow architecture*.

The dataflow concept is not new and during the 1970s and 1980s the language Lucid emerged. It is a comparatively small, pure and declarative language with an execution model that is structured around how data moves through the program instead of how the next step in the control sequence is reached. Parallelism is implicit in the language design, allowing the user to express the essence of the problem, as opposed to its efficient execution. The implementation will then work out how to achieve this latter part. During the 1980s the sequential pLucid interpreter was created as a major implementation of this language.

It might be that the version of Lucid described so many years ago (or a derivative thereof) will map nicely onto modern many core architectures but to answer this question we need, as a starting point, an implementation that is clear, modular, well documented and hence easily modifiable. pLucid is fast but it fails in all of these points as it was written in C and highly optimised at a very low level. It is my aim to construct a modern implementation of a Lucid interpreter from scratch, using a high level language to provide the required features.

Resources Required

- SVN Repository on the PWF filespace, so that I can work from various workstations
- latest stable version of Scala (currently 2.7.6)
- my PC (as workstation)
- Lucid book from the UL
- Scala book
- Optional: If native dataflow extensions are considered, then a CUDA capable GPU might be useful
- Linux development environment with standard tools (vim, svn and ssh clients, latex, etc...)

Starting Point

Prior Knowledge

Books:

- Odersky M., Spoon L., Venners B., 2007 *Programming in Scala* – I have read it completely and attempted some of the exercises
- Wadge W.W., Ashcroft E.A., 1985, *Lucid, the Dataflow Programming Language* – out of print, UL owns a copy, I currently have it on loan

Papers:

- Ashcroft E.A., Wadge W.W., 1984, *The Syntax and Semantics of Lucid*
- Plaice J., Mancilla B., Ditu G., 2008, *From Lucid to TransLucid: Iteration, Dataflow, Intensional and Cartesian Programming*

I have also gained some experience with Scala during a three week coding project of my summer internship.

To get a feel for the lazy semantics of Lucid I wrote a small program in Scala that implemented the classic, lazy *howfar* program using the actor based execution model. This worked and serves as a small proof-of-concept. The compiled jar, together with instructions to run it can be found on the project wiki (see below).

Software

- Operating Systems: Fedora Core 11 (64 bit), WinXP sp3 professional (32 bit)
- under both operating systems: sun JDK 1.6 or later, Scala 2.7.5.final
- windows only: successfully installed CUDA drivers and CUDA SDK, but haven't really used it yet

Hardware

- 2.6 GHz Dual Core Centrino notebook (64 bit architecture), 4GB RAM, GeForce 8400M GS gfx chip (CUDA capable)

Substance and Structure of the Project

The interpreter can be separated into two major sections, the frontend (FE) and the backend (BE), which can be developed independently. The FE is a parser that transforms an input program in the source language Lucid into an Abstract Syntax Tree (AST) that serves as the intermediate representation of the program. The BE is a runtime engine that then evaluates the parsed program.

The implementation language is Scala which satisfies the high-level criterion mentioned in the introduction and comes with some features and libraries that are particularly useful to the structure of my implementation. Additionally, Scala compiles to pure Java bytecode hence providing portability.

The FE consists of global datastructures like symbol tables, a lexer, a parser, and zero or more optional stages (see below for details). The lexer, parser and other stages will be arranged in a pipelined structure. When the parsing stage completes successfully, the input program will be represented in the form of an AST which serves as the intermediate format. Any further stages must consume

an AST as input and produce an AST as primary output. This allows easy composition. I will use the Scala Parser Combinator library, which merges the lexer and parser into a single system. Parser code written in this style closely mirrors the underlying grammar, so it is easy to comprehend what is going on and hence easy to change the grammar and thereby modify the accepted input language. This allows me to start with a simple subset of the final grammar and then extend it incrementally, without rewriting the system each time.

The BE consists of a textual user interface, a translation unit, a runtime engine and a framework that holds it all together. The engine is a highly idealized abstraction of a dataflow architecture where programs are represented as a network of computation nodes (as many as are required). These nodes communicate by asynchronously passing around requests for data items and corresponding replies. This layout is efficiently implementable on the Scala Actors library, Scala's native concurrency model. The translation unit takes an AST from the FE and creates such a network, possibly using an internal domain specific language (DSL) to encapsulate the core construction routines. This is again natively supported in Scala. The "loose" ends of these networks, which represent the input and output streams of the program, will be tied to the UI. When everything is set up, the system operates in a demand driven way, requesting the next value from the output which will trigger a ripple of request messages inside the network. When all data values have been provided on the inputs and all computations have completed a response with the result will reach the output. This implements the lazy evaluation semantics of Lucid.

I am keeping track of my progress on a project wiki page that is backed up daily. This allows me to store ideas that might be useful at a later stage as well as keeping a log of meetings and other developments related to the project:

`http://kudos.chu.cam.ac.uk/kwiki/index.php/Jonas_Kaiser`

Extensions

Here are some extensions that are not part of the core project, but that might enhance the usability, speed, or analytical capabilities of the interpreter.

- (general) performance comparison with pLucid, try not to fall too far behind in runtime behaviour
- (FE) a static type checker

- (FE) a pipeline stage that just analyses the AST and dumps some diagnostic data (e.g. a dataflow graph)
- (FE) other standard compiler optimisations
- (BE) extend computation nodes with local caches to reduce number of messages (warehousing)
- (BE) equip framework with diagnostic hooks to extract detailed runtime statistics
- (BE) if caches are used, add a simple garbage collector
- (BE) restrict the number of available nodes and cache sizes (i.e. getting closer to what hardware could provide natively)
- (BE) offload simple parts of the network to GPU

Success Criterion

For the project to be deemed a success the following items must be successfully completed.

- A correct interpreter accepting one-dimensional first-order Lucid (i.e. dimensions and functions are *not* considered as first order values) consisting of separated FE and BE
- A correct FE with a variable pipeline of at least two stages: Lexer and Parser.
- A correct BE that exhibits the lazy evaluation semantics of Lucid

For "correctness" tests I will use the old pLucid interpreter as a reference by using the provided example programs. In some areas my interpreter will be less expressive than pLucid, so programs using the respective features have to be excluded (e.g. multidimensionality).

Timetable and Milestones

The following list is the division of the available time into 10 work slots of two weeks each, together with major objectives that I want to tackle in each of these periods. Each slot also has one or more milestones that should be achieved by the end of the respective period. The longer periods over Christmas and Easter will allow me to catch up if something is running late or to gain a head start if there is sufficient time.

Slot 1: 24th October – 6th November

Preparation and Research Phase:

- Read Lucid Book
- Read several Lucid papers
- Research: Scala Actors library
- Research: Scala Parser Combinator library; Write a parser for simple arithmetic expressions
- First version of AST
- Choose test programs for zero order Lucid
- Create supporting datastructures for the FE
- First version of parser (targeting zero-order Lucid – only the very basics, no functions)

Milestones: Running "Hello World" parser – Working FE for zero order Lucid – Written record of research results

Slot 2: 7th November – 20th November

Coding phase 1:

- First version of dataflow network implementation (zero order features only)
- First implementation of the translation unit

- Clean UI
- Tie zero order FE and BE together to produce first stable interpreter
- Correctness Testing

Milestone: Working Interpreter for zero order Lucid

Slot 3: 21st November – 4th December

Coding phase 2:

- Choose test programs for first order Lucid (user functions and `is current` operator)
- Extend AST
- Extend parser to accept first order Lucid
- Implement first order features in the BE
- Adjust translation unit to accommodate changes
- Tie first order FE and BE together to produce second stable interpreter
- Correctness Testing

Milestone: Working Interpreter for first order Lucid

Christmas break

Before Christmas:

- Finish Milestones that haven't been achieved yet
- Major Debugging + Correctness Testing
- Review whether any extensions are possible
- Start Michaelmas revision
- Vacation

After Christmas:

- Compose ideas for the dissertation, write fragments of text, etc ...
- Start implementation of first order features for the BE

Slot 4: 9th January – 22nd January

Coding phase 3:

- Continue Major Debugging + Correctness Testing
- Start on Progress Report

Milestones: Stable First Order Lucid Interpreter – Draft Progress Report

Slot 5: 23rd January – 5th February

Evaluation phase 1:

- Complete Progress Report and practice presentation
- Improve system documentation
- Document / Demonstrate ease of system composition; start with identity transform, then make it more interesting
- Plan which extensions can be achieved during slots 6 and 7, focus on pLucid performance comparisons

Milestones: Finished Progress Report on 29th January, 12am noon – Presentation – System Documentation – Detailed plan for slots 6 and 7

Slot 6: 6th February – 19th February

Evaluation phase 2:

- Perform work planned earlier, e.g. set up and run performance tests
- Document results and incorporate conclusions where possible

- Repeat bug fixes and correctness tests where necessary due to changes

Milestones: Interpreter is still stable – Collected large section of dataset as specified in plan from previous stage

Slot 7: 20th February – 5th March

Evaluation phase 3 / Dissertation phase 1:

- Perform any uncompleted tests left from previous stage
- Review, consolidate and organize obtained data set
- Set up dissertation, import any fragments already written along the way, start work on chapters 1 to 3 properly

Milestones: Written record of completed evaluation data set – Full draft of *Introduction*, *Preparation* and *Implementation*

Slot 8: 6th March – 19th March

Dissertation phase 2:

- Finish Chapters 1 to 3
- Start work on Chapter 4
- Verify formal constraints, layout, support sections

Milestones: C1 - C3: final draft – *Evaluation*(C4): first full draft

Easter Break

- Polish Chapter 4, create and place diagrams, graphs, etc.
- Create an initial version of Chapter 5 *Conclusion*
- Compose first draft of dissertation and send to Supervisor and DoS
- Revise Lent term, work out which courses are good exam options

Milestone: First draft of Dissertation

Slot 9: 17th April – 30th April

Dissertation phase 3:

- Finish Chapters 4 and 5 and incorporate feedback
- Compose Appendix, TOC, Bib, index
- Assemble final dissertation
- Start major proof reading

Milestones: Final draft of Dissertation – performed and processed several Proof readings

Slot 10: 1st May – 14th May

Dissertation phase 4:

- Complete proof reading
- Compose final version of Dissertation
- Double check commission procedure
- Get it printed and bound
- Hand in before deadline

Milestone: Hand everything in on time, final deadline 14th May, 12:00am noon