Computer Science Tripos Part II

Interpreting a Declarative, Parallel Structured Grid Computing Embedded Domain-Specific Language

May 11, 2011
Original Aims of the Project

Ypnos is a declarative, parallel structured grid computing embedded domain-specific language, which is currently embedded in Haskell. I intend to extend the HaMLet interpreter in order to embed Ypnos in ML. An Ypnos program consists of a many-dimensional grid of elements and a stencil function which is applied to each element. A system of grid patterns is used to declare which elements a stencil function uses as its arguments, relative to the cell being written to. Parallelisation of Ypnos programs is performed by the interpreter in a predictable way. This is achieved by enforcing the single, independent writes property.

Work Completed

I have completed all proposed criteria and two proposed extensions. I extended an open-source ML interpreter to embed a parallel, domain-specific language. I have completed an extension, which was not initially proposed,
to build a compiler as well as an interpreter. My compiler is 15,000 times faster than the existing prototype implementation of the language.

Special Difficulties

None
Declaration

I Oliver R. A. Chick of Gonville & Caius 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 May 11, 2011
# Contents

1 Introduction ...................................................... 1
  1.1 Ypnos .......................................................... 1
  1.2 Structured Grids ............................................... 2
  1.3 Stencil Functions .............................................. 3
  1.4 Parallelisation ............................................... 4
  1.5 Project Summary .............................................. 4

2 Preparation ....................................................... 5
  2.1 Prerequisite Knowledge ....................................... 5
  2.2 Software Engineering Techniques ............................ 6
  2.3 Requirements Analysis ....................................... 6
  2.4 HaMlet .......................................................... 7
  2.5 Parallel ML .................................................... 7
  2.6 Category Theory ............................................... 8
    2.6.1 Monads ................................................... 9
    2.6.2 Comonads ................................................. 10
    2.6.3 Operations on Grids ...................................... 12
  2.7 Haskell ........................................................ 12

3 Implementation .................................................... 13
  3.1 Overview ..................................................... 13
  3.2 Interpreting and Compiling YpnosML ....................... 14
    3.2.1 The Frontend ........................................... 15
    3.2.2 Domain Decomposition .................................. 17
    3.2.3 Concurrency Mechanisms ................................ 18
    3.2.4 Interpreting YpnosML .................................. 19
    3.2.5 Compiling YpnosML .................................... 20
  3.3 Language Restrictions ....................................... 24
  3.4 Default Values ............................................... 26
  3.5 Making YpnosML Practical .................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>Reductions</td>
<td>28</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Unzip</td>
<td>29</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Fmap</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>The Definition of Ypnos</td>
<td>30</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Grammar</td>
<td>31</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Semantics</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Experimental Setup</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Evaluation Algorithms</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Image Converters</td>
<td>39</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Objectives</td>
<td>39</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Evaluation Hardware</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Ypnos Correctness</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison between YpnosML and YpnosH</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Speed</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Memory Footprint</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Parallelism</td>
<td>44</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Game of Life</td>
<td>46</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Gaussian Blur</td>
<td>47</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Edge Detection</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison with Optimised C</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>Future Performance Improvements</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Overview</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Improvements</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Overview</td>
<td>57</td>
</tr>
<tr>
<td>A</td>
<td>Compiled Ypnos Gaussian Blur</td>
<td>59</td>
</tr>
<tr>
<td>B</td>
<td>Gaussian Blur Written in C</td>
<td>65</td>
</tr>
<tr>
<td>C</td>
<td>Project Proposal</td>
<td>73</td>
</tr>
<tr>
<td>C.1</td>
<td>Introduction and Description of the Work</td>
<td>73</td>
</tr>
<tr>
<td>C.2</td>
<td>Resources Required</td>
<td>74</td>
</tr>
<tr>
<td>C.3</td>
<td>Starting Point</td>
<td>74</td>
</tr>
<tr>
<td>C.4</td>
<td>Substance and Structure of the Project</td>
<td>75</td>
</tr>
<tr>
<td>C.5</td>
<td>Variants</td>
<td>76</td>
</tr>
<tr>
<td>C.6</td>
<td>Success Criteria</td>
<td>77</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>C.7</td>
<td>Extensions</td>
<td>77</td>
</tr>
<tr>
<td>C.8</td>
<td>Timetable and Milestones</td>
<td>78</td>
</tr>
<tr>
<td>C.8.1</td>
<td>Preliminary Work</td>
<td>78</td>
</tr>
<tr>
<td>C.8.2</td>
<td>Weeks 1 and 2</td>
<td>78</td>
</tr>
<tr>
<td>C.8.3</td>
<td>Weeks 3 and 4</td>
<td>78</td>
</tr>
<tr>
<td>C.8.4</td>
<td>Weeks 5 and 6</td>
<td>78</td>
</tr>
<tr>
<td>C.8.5</td>
<td>Weeks 7 and 8</td>
<td>78</td>
</tr>
<tr>
<td>C.8.6</td>
<td>Weeks 9 and 10</td>
<td>79</td>
</tr>
<tr>
<td>C.8.7</td>
<td>Weeks 11 and 12</td>
<td>79</td>
</tr>
<tr>
<td>C.8.8</td>
<td>Weeks 13 and 14</td>
<td>79</td>
</tr>
<tr>
<td>C.8.9</td>
<td>Week 15 to 20</td>
<td>79</td>
</tr>
</tbody>
</table>

**Bibliography**  

81
Chapter 1

Introduction

Many scientific computing problems require parallelisation because they involve too much data to be performed efficiently on a single core processor. Existing languages largely use one of two approaches perform parallelisation: a manual approach that requires the programmer to annotate their code, increasing complexity, or a compiler-driven approach with complex analyses, however this approach can create unpredictable runtime memory usage and execution times. The non-deterministic nature of parallel code significantly increases the difficulty of debugging.

Ypnos [1] is a parallel embedded domain-specific language (EDSL) which uses a compiler-driven, annotation-free approach to create predictable parallelised code. An Ypnos prototype has been embedded into Haskell. I embed Ypnos into Standard ML (SML), expanding the utility. I present the computational pattern and theoretical background behind Ypnos, its grammar and semantics and analyse the performance of both the SML and Haskell versions.

1.1 Ypnos

Ypnos is the abstract concept of an EDSL designed to produce predictable, parallel code without requiring programmers to annotate their code. One implementation of Ypnos is embedded into Haskell; I refer to this as Ypnos\textsubscript{H}. This work embeds Ypnos into SML; I refer to this as Ypnos\textsubscript{ML}.

Ypnos performs computations on structured grids, a computational pattern commonly used in graphics, finance and other scientific computations. A stencil function is applied to each element of a many-dimensional, array-like
data structure. The stencil function computes the next value of each element, based on the current element and its neighbouring values.

As an embedded domain-specific language, Ypnos provides domain specific expressivity in an inexpensive way. It is able to take advantage of the features of a host language whilst offering a declarative, functional approach for parallelising structured grid computations. Although other languages exist that parallelise structured grid computations, Ypnos:

- Uses simple, static analysis, to create programs whose runtime costs are predictable.
- Does not require manual parallelisation or annotations, unlike C with MPI [2].
- Guarantees correct parallelisation by not using side-effects.
- Has convenient syntax to reduce the length of programs.

1.2 Structured Grids

A grid is a many-dimensional, array-like data structure. Figure 1.1 shows a grid of dimension \(5 \times 5\) and element type \(\text{int}\). The grid has type

\[
\text{Grid} (X \times Y) \text{ int}
\]

and is like the Java type \(\text{int}[][]\).

In Ypnos, a grid of dimensions \(D\) and elements of type \(\alpha\) has type:

\[
\text{Grid} D \alpha
\]

Figure 1.2 shows a simplified Java structured grid computation averaging a cell’s neighbours. In reality this code would require correct handling of cases at the edge of the array.
1.3. STENCIL FUNCTIONS

Figure 1.2 An example program showing a sequential structured grid computation as written in Java

```java
for (int i=0; i<ROWS; i++)
    for (int j=0; j<COLS; j++)
                        + grid[i-1, j+1] + grid[i, j+1] + grid[i+1, j+1]) / 9;
grid = tmpGrid;
```

1.3 Stencil Functions

A **stencil function** is a function from a structured grid to a value.

\[ \text{Grid } D \alpha \rightarrow \beta \]

Ypnos stencil functions consist of a **grid pattern**, a declarative way of binding values to variables and a **functional expression**, usually containing these bound variables. A grid pattern contains a number of variables which are bound to elements of a grid, based on their lexical position to a **focal point**.

For example in the grid pattern

```
| tl tl c tr |
| @ @ @ c r @ @ |
| @ @ b l b c b r |
```

the variables \( l, c \) and \( r \) are bound to three consecutive elements of a grid. \( tl, tc \) and \( tr \) are bound to the elements above \( l, c \) and \( r \) respectively. The `@@` annotation indicates that \( c \) is the focal point of the stencil function.

The **run** function in Ypnos binds each element of a grid to the focal point and repeatedly runs the stencil function in parallel, producing a new grid.

\[ \text{run} : (\text{Grid } D \alpha \rightarrow \beta) \rightarrow \text{Grid } D \alpha \rightarrow \text{Grid } D \beta \]

Stencil functions provide Ypnos with a compact and elegant notation. The interpreter handles the complexities of parallelisation. For example Figure 1.3 is a complete YpnosML program that averages the value of each element and its neighbours. It is worth comparing for clarity with Figure 1.2, showing a similar and simplified program in Java.
CHAPTER 1. INTRODUCTION

Figure 1.3 A stencil function and run expression for computing the average of a grid cell and its neighbours.

YPNOS \((X \ast Y)\):

\[
\begin{array}{cccc}
| & a & b & c | \\
\hline
@@ | d & @@ & e & f | @@
\hline
| g & h & i | \\
\hline
\end{array}
\]

\[= \Rightarrow (a + b + c + d + e + f + g + h + i)/9.0\]

1.4 Parallelisation

Ypnos is an annotation-free parallel language. The programmer can specify their problem in a concise and less error-prone way. To ensure programs are correct when parallelised, Ypnos enforces the single independent writes (SIW) property: each application of a stencil function makes a single write to the focal point to guarantee that writes do not overlap.

The ethos, of abstracting from the details of parallelisation, is extended to the runtime cost of Ypnos. The SIW property allows the compiler to parallelise programs by domain decomposition of grids. This compile time analysis creates code that has few concurrency controls, so parallelisation is cheap.

Ypnos is not tied to specific hardware. Possible implementations are to use kernel threads on a chip multiprocessor, distributing the computation over a cluster or performing on a GPU. In this project, I have a backend of kernel threads running on a chip multiprocessor.

1.5 Project Summary

In this work, I present my YpnosML interpreter and YpnosML compiler—an extension that was not initially proposed. The interpreter and compiler include all my initial success criteria and two of my proposed extensions: type inference and grid reductions.

In Section 4, I show YpnosML is 15,000 faster than YpnosH, making YpnosML the fastest implementation of Ypnos. I also show YpnosML performance gains of up to 50% from parallelism using sixteen cores. I compare YpnosML to C, showing that whilst C programs can run twice as fast as YpnosML programs, further work could reduce this gap.
Chapter 2

Preparation

Before implementing Ypnos\textsubscript{ML}, I considered the software engineering approach to use to handle the complexity of writing an interpreter and compiler. Much of the work draws on knowledge from the Computer Science Tripos and other subjects, notably \textit{category theory}. I had to learn the organisation of HaMLet \cite{3}, the SML interpreter that I extended and the \textit{Threads} structure in Poly/ML \cite{4}.

2.1 Prerequisite Knowledge

To build Ypnos\textsubscript{ML}, I used knowledge from the following courses:

\textbf{Compiler Construction} to understand the structure of a compiler and tools involved in extending HaMLet.

\textbf{Concepts in Programming Languages} to understand the SML module system, which HaMLet uses.

\textbf{Concurrent and Distributed Systems} to build concurrency controls into Ypnos\textsubscript{ML} to ensure thread safety, also to correctly parallelise SML features.

\textbf{Comparative Architectures} to understand the underlying hardware and the limits of parallelism.

\textbf{Computer Graphics and Image Processing} to create programs for the evaluation.
Semantics of Programming Languages to write the semantics of Ypnos_{ML} and understand *The Definition of Standard ML*, which was necessary to understand HaMLet.

Software Engineering to use appropriate techniques to organise the project.

Types to extend the SML type system.

2.2 Software Engineering Techniques

It is important that an interpreter and compiler be reliable, since programmers do not want bugs introduced because of these tools. I considered the software engineering approach that would be most appropriate to my project and time frame.

I use the *spiral model* because some requirements could not be set out initially. These included the required performance and extensions that would be included. I regularly met with my supervisor and the designer of Ypnos to discuss the implementation and features required. These meetings led to me extending HaMLet with compiler support, due to performance issues.

Care must be taken to ensure that an interpreter or compiler correctly evaluates all code. To ensure Ypnos_{ML} is reliable, I wrote a program for each feature to compare the result of that program with an expected result. Every test was run each time I added a new feature to prevent *regression bugs*.

I used *Git* [5], a version control system, to track changes and revert any that introduced regression bugs. All code was replicated to three backed-up servers.

2.3 Requirements Analysis

Ypnos was designed to improve parallel computing, specifically for structured grid computations. The language concept allows multiple implementations as programming languages. The only prototype of Ypnos is Ypnos_{H}. The aim of my project was to embed a version of Ypnos into SML to perform an evaluation between the two implementations and confirm that the concept can be ported between languages.

To embed Ypnos in SML, I produced the following specific objectives:
1. An implementation of an \((X \times Y)\) grid data structure representation of a finite 2-dimensional discrete data structure.

2. An internal representation of a valid Ypnos stencil function must be designed and implemented.

3. A process of lexing and parsing Ypnos grid patterns.

4. The interpreter must apply a stencil function to a grid and produce another grid containing the correct output.

5. The solution must produce parallelised code which runs on a variable number of processors.

6. A detailed evaluation of the efficiency in parallelisation of the interpreter must be performed using up to 16-cores.

2.4 HaMLet

As an embedded language, YpnosML requires a SML interpreter that can be extended to support the extra features. Building one from scratch would have required too much work and been unstable. Instead, I chose to extend an open-source ML interpreter, HaMLet [3]. HaMLet was designed to be a tool for ‘experimentation with the ML semantics or extensions to it’ and is well-documented. HaMLet is a close implementation of The Definition of Standard ML [6] with no optimisations. Before beginning the project, I read through the source code of HaMLet. The ease in which I could learn how HaMLet worked made it an excellent choice. However, the lack of optimisations made HaMLet slow and caused performance problems.

Figure 2.1 shows the structure of the HaMLet interpreter.

2.5 Parallel ML

Ypnos was designed to be a parallel EDSL so any implementation should create parallel code. I chose to support parallelisation by using threads on a chip multiprocessor. This simplifies the interpreter, as it uses a shared memory model rather than a distributed architecture, eliminating the need to decompose Ypnos programs over a distributed topology.
The Definition of Standard ML [6] does not specify any parallel semantics, so there is no standard method of spawning threads. The only mainstream ML compiler that supports concurrent threads on different cores is Poly/ML [7]. Poly/ML’s threads are based on the pthread package: threads can be spawned by calling fork and synchronisation is supported by using mutexes and condition variables [4].

HaMLet, compiled with Poly/ML, can include calls to spawn threads. If this functionality is used in the evaluation module of the interpreter, the backend can use threads whilst the programs accepted must be single threaded. This is ideal for Ypnos, as Ypnos programs should not explicitly spawn threads.

### 2.6 Category Theory

Category Theory formalises the Ypnos grids and the denotations of the operations that are performed on them. Orchard, Bolingbroke and Mycroft [1] use category theory to introduce YpnoSH.

SML is an impure language with support for exceptions, state and side-effects, however Ypnos was designed to be used as a pure language. The advantage of Ypnos being pure is that it can give a stronger guarantee of correctness when programs are parallelised. If impure functions are used in YpnoSM, a race condition might exist whereby side-effects are triggered in a different order each time the program is run, due to the thread scheduler. This non-determinism is undesirable.
In YpnosML the type system prevents the programmer from using ref cells in grids. Otherwise, impure stencil functions and grids are permitted but their use is discouraged. However, side-effects are a useful programming construct, which many programmers will use. By using category theory, a programmer can write pure Ypnos programs that encapsulate side-effects, achieving both purity and the ease of programming. Category theory is an abstract way of considering objects and the relationships between them, morphisms.

2.6.1 Monads

Consider the functions:

- \( \text{div} : (\text{Real} \times \text{Real}) \rightarrow (\text{Real} + 1) \): may throw a division-by-zero exception.
- \( \text{print} : (\text{String} \rightarrow \text{IO}()) \): returns () and prints to the screen.
- \( \text{setState} : u \rightarrow \text{parseState} u () \): returns () and changes the program’s state.

All these functions have a possible side-effect which YpnosML should allow using only pure programming constructs. To represent a function \( A \rightarrow B \) that can cause a side-effect in a pure way, we can use a monad [8]. Functions of type \( A \rightarrow B \) are replaced with functions of type \( A \rightarrow T B \), where \( T \) is a monad.

A monad is defined as a triple \((T, \eta, \mu)\) [9]:

**Underlying Functor** \( T \) is an endofunctor; a functor from a category \( C \) to itself, \( C \rightarrow C \).

**Unit** \( \eta : \text{Id}_C \rightarrow T \). Unit turns a value into a monad. Unit can be applied to a monadic value. If a value \( A \) has a monad \( T \) applied to it \( n \) times, the result is \( T^n A \).

**Multiplication** \( \mu : T^2 \rightarrow T \). Multiplication reduces the number of times a monad, \( T \) is applied to a value by one.

The operations \( \eta \) and \( \mu \) must obey the following constraints:

\[
\begin{align*}
TTA & \xrightarrow{\eta T A} TTA & \xrightarrow{T TTA} TTA & \xrightarrow{T TTA} TTA \\
TTA & \xrightarrow{\mu_A} TA & \xrightarrow{T \mu A} TA & \xrightarrow{\mu_A} TA
\end{align*}
\]

Monads are commonly used in Haskell; however, there is no built-in support for monads in SML. Instead, monads can be implemented in SML by a
programmer, using a polymorphic type constructor. For example, to write an exception monad:

```ml
exception e of string

datatype α T = Raise of exn
| Return of α

fun unit a = Return a : α → M α

fun join m k = case m of
  Raise e => e
| Return a => a : T(T α) → T α
```

The endofunctor, $T$, is a pure datatype that can encapsulate either the result of a computation or an exception type. The unit function will take a variable and build a monad containing that variable. Join will remove one monad from any argument. The monad can be used in writing an ML function with an exception side-effect, for example `divide`:

```ml
fun divide (a,b) = if b=0
  then raise ("divide by zero")
  else unit (a div b)
```

The use of monads allows pure functions to throw an exception. This function is appropriate for use in YpnosML, whereas a function using SML’s impure side-effects is not.

### 2.6.2 Comonads

Structured grids are the crux of Ypnos: all Ypnos computations are centred around applying stencil functions to grids. The grid models another part of category theory, the *comonad* [10]. Comonads are much less widely understood and used than monads, however they have some similar properties.

The *Lucid* programming language [11] describes streambased dataflow equations. Lucid is a language that performs a series of *context-dependent computations*—computations that map a *context* to a value. The later analysis of Uustalu and Vene [12, 13] show that Lucid can be formulated in terms of comonads. This inspired Ypnos designers to consider structured grid computations in a similar way.
2.6. CATEGORY THEORY

Consider the following functions:

Next item in a stream \textit{next} : Stream \( \alpha \to \alpha \)

Staged computation \textit{eval} : \( \Box \alpha \to \alpha \)

Stencil function \textit{stencil} : \((\text{Grid} \; \alpha \times i) \to \beta\)

All of these functions are mappings from a context, such as a stream or a grid, to a value. Comonads are the dual of a monad, representing functions with context. This can be expressed by the type

\[ f : D a \to B \]

where \( D \) represents a comonad.

A comonad is defined by a triple \((D, \varepsilon, \delta)\):

Underlying Functor \( D \) is an endofunctor, from a category \( C \) to itself, \( C \to C \).

Counit \( \varepsilon : D \to ID_C \) Counit makes a comonad context-independent, by discarding context.

Comultiplication \( \delta : D \to DD \) given a comonad, comultiplication will add more context to the comonad, building a comonad of the supplied comonad.

The operations \( \varepsilon \) and \( \delta \) must obey the following conditions:

\[
\begin{array}{c}
DA \xrightarrow{\delta_A} DDA & DA \xrightarrow{\delta_A} DDA \\
\delta_A & \delta_A \\
\varepsilon_{DA} & D\varepsilon_A \\
DDA \xrightarrow{\varepsilon_{DA}} DA & DDA \xrightarrow{\delta_{DA}} DDDA
\end{array}
\]

A grid parameterised with its dimensions can be represented as a comonad. For instance \textit{Grid} \((X \times Y)\) is a comonad. Any stencil function applied to a grid is an instance of counit. A stencil function takes a context—the grid and the focal point—and performs a computation based on this context. The most simple instance of a counit is shown in figure 2.2.

Figure 2.2 Source code and diagrammatic representation of counit in Ypnos

\begin{verbatim}
val counit = YPNOS (X*Y) : @ @ | @@x | @ @ => x
\end{verbatim}

Comultiplication, \( \delta : \alpha \text{ grid} \to (\alpha \text{ grid}) \text{ grid} \) will build a grid of grids.
2.6.3 Operations on Grids

I have explained how category theory represents side-effects with monads and model a grid with comonads, however this theory does not yet model the \texttt{run} operation. To model \texttt{run}, more operations from category theory are required:

\begin{align*}
\text{\texttt{fmap}} & : (\alpha \rightarrow \beta) \rightarrow (D\alpha \rightarrow D\beta). \text{ \texttt{fmap} lifts a function to apply it to each element of a comonadic data structure.} \\
\text{\texttt{cobind}} & = (\text{\texttt{fmap}} \circ \delta) : (D\alpha \rightarrow \beta) \rightarrow (D\alpha \rightarrow D\beta). \text{ \ Takes a function from a comonad, } D\alpha \text{ to a value, } \beta.
\end{align*}

The \texttt{run} operation of Ypnos is the \texttt{cobind} operation: it lifts a stencil function over a comonadic data structure—the grid.

2.7 Haskell

There is an existing prototype implementation of Ypnos embedded in Haskell: a lazy, pure, functional language. I had to be able to use Ypnos\textsubscript{H} to compare its runtime performance with Ypnos\textsubscript{ML} so I had to learn Haskell. Although the syntax of Ypnos\textsubscript{H} and Ypnos\textsubscript{ML} is similar, Ypnos\textsubscript{H} is embedded into Haskell using quasi-quoting [14]. Also the syntax of the right-hand side of a stencil function in Ypnos\textsubscript{H} is written as a Haskell expression.
Chapter 3

Implementation

3.1 Overview

Many scientific computations process large amounts of data, so require a large amount of processing power to be computed in a reasonable amount of time. Typically, a programmer uses thread-level and data-level parallelism to exploit a multicore processor and improve performance. Most existing parallel languages, such as C+OpenMP [15] require the programmer to annotate their code to perform this parallelisation, which is complex and error-prone. Other languages use compiler-driven approaches to create parallel code, which can result in unpredictable runtime costs.

YpnosML is my implementation of Ypnos, an embedded language that uses a simple static analysis to produce parallel code without annotation. My prototype implementation lets programmers specify structured grid computations, with a concise syntax that produce parallel code. YpnosML can both interpret and compile Ypnos code1.

YpnosML is domain-specific to the structured grid computation pattern and lets programmers use grids as a SML datatype. Grid represents a two-dimensional structured grid, paramaterised by its dimensions and element type. YpnosML uses type-inference to detect the element type but requires the programmer to explicitly declare the dimensions, which are checked.

YpnosML supports stencil functions, using Ypnos’s grid pattern syntax and a SML functional expression, as shown in Figure 3.1. This stencil function

1Compiling Ypnos is an extension to the initial project proposal; added to improve the performance of YpnosML programs.
Figure 3.1 The stencil function of the Sobel operator edge detection algorithm written in YpnosML.

```plaintext
val sobel = YPNOS (X*Y) : | a b c |
    @@ d @@ f @@
    | g h i | =
Math. sqrt (Math.pow(1*a+c-2*d+2*f-g+i, 2.0) +
Math.pow(a+2*b+c-g-2*h-i, 2.0))
```

consists of:

**Grid pattern** on lines 1–3 assigns values to variables \([a..i]\) based on their lexical location to the element bound to the focal point, \(e\).

**ML expression** on lines 4–7 uses the values of the bound variables in a SML-expression. This expression will convolve \(L_x\) with one element of the grid, \(G\) and convolve \(L_y\) with the same element of \(G\). When this stencil function is applied to run, \(L_x\) and \(L_y\) will be convolved with the entirety of \(G\).

\[
L_x = \begin{bmatrix}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1 \\
\end{bmatrix}
\]

\[
L_y = \begin{bmatrix}
+1 & +2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{bmatrix}
\]

From these convolutions the gradient magnitude is computed as:

\[
|\Delta L| = \sqrt{(L_x * G)^2 + (L_y * G)^2}
\]

YpnosML, like YpnosH, only supports two-dimensional grids and patterns. YpnosML programmers can build grids of grids, to represent many-dimensional grids. Extending grid patterns to many dimensions is syntactically difficult.

### 3.2 Interpreting and Compiling YpnosML

YpnosML is a port of the design ideas of Ypnos to SML. I have extended HaMLet to produce a fully functional YpnosML interpreter and compiler.
3.2. INTERPRETING AND COMPILING YPNOS\textsubscript{ML}

3.2.1 The Frontend

Lexing and Parsing

Ypnos\textsubscript{ML} extends HaMLet’s lexer, which is written in ML-Lex [16]. ML-Lex is a program that generates a lexical analyser from a grammar, based on Lex [17]. It can be considered as a function:

\[
\text{Grammar keywords} \rightarrow \text{Character stream} \rightarrow \text{Token stream}
\]

I extended HaMLet’s parser, written in ML-Yacc [18], to reflect the Ypnos additions. ML-Yacc is a conventional LALR(1) parser generator, based on Yacc [19]. It takes a description of the grammar and produces a parser that converts a token stream into an abstract syntax tree.

\[
\text{Grammar description} \rightarrow \text{Token stream} \rightarrow \text{Abstract syntax tree}
\]

For the lexer and parser, my plan was to extend individual modules one at a time and test them. For example, extending the lexer to include the features of Ypnos and then extending the parser and then type checking. I varied from this plan due to software dependencies in the organisation of HaMLet. To modify the lexer, the datatypes have to be updated. There are dependencies from most files in HaMLet to these data structures so I had to update nine other files. Discovering these dependencies to make even simple changes to HaMLet was time consuming: if all the files are not updated then HaMLet can still be compiled, however it outputs ‘Undetermined types at top level’ whenever any ML code is entered and segfaults. HaMLet cannot be compiled with debugging enabled so tracking this bug had to be done manually. As HaMLet contains 80,000 lines of code, it was difficult to find all the files that had to be modified.

Dependent Type Checking

HaMLet performs type checking and type inference in the elaboration stage. The elaboration of an Ypnos\textsubscript{ML} program determines if given \(\Gamma\) and \(M\) there exists a \(\tau\) such that:

\[
\Gamma \vdash M : \tau
\]

If \(\tau\) exists then the program is type correct, otherwise the program is rejected. Elaboration works on the abstract syntax tree, producing a typed abstract
CHAPTER 3. IMPLEMENTATION

syntax tree.

elaboration : abstract syntax tree → typed abstract syntax tree

YpnosML extends the HaMLet type inference algorithm, which is based on the Hindley–Milner algorithm [20]. YpnosML programs verify that all of the rows of a grid are the same size and that the dimensions match the declared dimensions. For example both of the following grids are invalid:

\[
\text{Grid}(2 \times 2) \begin{array}{c}
1,2
\end{array} \begin{array}{c}
3,4,5
\end{array}
\]

\[
\text{Grid}(3 \times 3) \begin{array}{c}
1
\end{array}
\]

To ensure grids are valid, dependent types [21] have to be used in the elaborator. A dependent type is a type that depends on a value. In YpnosML, the type of a grid depends on its dimensions. The Ypnos grammar specifies that the dimensions of a grid must be explicitly declared. The number of elements in each row must match the declared width and the number of rows must match the declared height, specified as

\[
\Pi n : \mathbb{N}, m : \mathbb{N}. \text{Grid}(n, m, \alpha)
\]

The length of each row is dependent on the length of all other rows and the declared row length. If these quantities do not match then the grid is invalid.

\[
\text{Grid}(n \times m) \begin{array}{c}
\alpha^n
\end{array} / / m
\]

Dependent types are implemented by passing the grid dimensions through the environment, \(\Gamma\), as shown in Section 3.6.2.

If an YpnosML program is typeable then it is believed, without being formally proven, to be sound [22]. That is if

\[
\Gamma \vdash M : \tau
\]

tythere exists no sequence of computation such that

\[
< M, \{} \rightarrow \ldots \rightarrow \text{FAIL}
\]

As YpnosML programs are believed to be sound, the type information is removed after elaboration and the typed abstract syntax tree is evaluated. The result of the evaluation is then combined with the result of the elaboration stage to generate a typed variable.
3.2. INTERPRETING AND COMPILING YPNOSML

Figure 3.2 Domain decomposition splits a grid into subgrids. A thread applies the stencil function on each subgrid. Subgrids are combined to form a new grid.

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Stencil application</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
<tr>
<td>a a a a</td>
<td>a b b b b</td>
<td>b b b b b</td>
</tr>
</tbody>
</table>

3.2.2 Domain Decomposition

The typed abstract syntax tree is passed to the evaluation module of HaMLet to be interpreted. The evaluation module parallelises the tree, using domain decomposition, a simple, static analysis. Domain decomposition allows Ypnos to split a grid into smaller subgrids and assigns a thread to compute each subgrid, as shown in Figure 3.2.

YpnosML enforces the SIW property on all operations: when building a new grid, each element must be written exactly once and its value is independent of the order in which it is processed. The SIW property guarantees the correct value is returned when a pure program is parallelised, independently of domain decomposition.

Domain decomposition exploits processor architecture to maximise performance. Modern processors typically have an L_1 and L_2 cache per core and a shared L_3 cache, which provide access to recently-read memory locations in 1–10 cycles of latency [23]. If an address is requested that is not cached it can take 400 clock cycles to fetch from main memory. YpnosML needs to maximise its cache hit rate to boost runtime performance.

Subgrids are statically calculated and assigned to each thread. Each core typically has a private L_1 and L_2 cache, so part of each subgrid is stored on the core’s private caches. Each iteration of a parallel function computes each subgrid on the same core. When many iterations are performed on a grid, the result of iteration \( n - 1 \) is likely to be cached on iteration \( n \).

YpnosML performs domain decomposition by splitting the grid along the shortest dimension that is larger than the number of threads. For example a grid of dimensions 50 \( \times \) 100 running on 8 threads will be split along the first dimension, i.e. 50, see Figure 3.3. By splitting the grid along the shortest
dimension, YpnosML maximises the probability that when a worker thread has computed the result of one row and moves onto the next, many of the elements will still be in the cache so only the new row causes cache misses.

Elements of a grid are stored in the order $[0, 0], [0, 1], \ldots, [0, n-1], [1, 0], \ldots, [m-1, n-1]$. On modern processors, when element $[i, j]$ is cached, its next consecutive elements are also loaded. YpnosML exploits this by processing grids row-by-row to maximise the chance of an addressed location already residing in a cache. When there is a cache miss, eight of the next values in the same row will be fetched into the core’s cache. Therefore, on the next computation, it is unlikely there will be another cache miss. If subgrids were processed vertically then the caches would still contain values in nearby columns however would not store the values of lower rows so the cache miss rate would be higher.

### 3.2.3 Concurrency Mechanisms

Domain decomposition allows YpnosML programs to be parallelised statically with few concurrency controls. YpnosML threads are independent of all other threads so there is little inter-thread communication. Concurrency controls and inter-thread communication are significant overheads of a thread, especially as Poly/ML threads are lightweight, so YpnosML uses these sparingly.

Whilst a thread processes a subgrid, it is independent of all other threads and has no communication with them. The thread writes to a new subgrid, which no other thread can access, so no concurrency controls are needed. It is only when a thread has completed processing a subgrid that it requires concurrency control.
When a parallel function is called, Ypnos\textsubscript{ML} spawns worker threads to compute each subgrid, whilst the main thread blocks until all the worker threads terminate. When the main thread wakes it returns the new grid. This mechanism makes all parallel functions synchronous, which is essential to the abstraction from parallelism Ypnos provides.

Each parallel function call generates a single mutex, which is initially locked. This mutex protects a counter storing the number of active threads. The main thread builds an empty grid of the correct dimensions and element type, which it divides into subgrids. Worker threads are spawned to compute each subgrid then the main thread waits on a condition variable, unlocking the mutex. As each worker thread terminates it locks the mutex, decrements the counter and then unlocks the mutex. The last worker thread signals the condition variable to wake the main thread, which returns the new grid.

Having low inter-thread communication allows Ypnos\textsubscript{ML} worker threads to use asynchronous thread communication, to reduce the overhead of each thread for CPU intensive tasks.

### 3.2.4 Interpreting Ypnos\textsubscript{ML}

As well as performing domain decomposition, the Ypnos\textsubscript{ML} evaluation module produces grids, runs parallel functions and maps variables used in grid patterns to elements of a grid.

Interpreted Ypnos\textsubscript{ML} extends the datatypes in HaMlet to include a grid structure that is parameterised by its element type. The interpreter backend uses a \texttt{VIdMap} to store grids, which is the same data structure used to store vectors, tuples and lists. The \texttt{VIdMap} provides a mapping from an ordinal key to a value. In the case of a grid, the key is the grid co-ordinates and the value is the element.

The functions used are direct translations from \textit{The Definition of Ypnos\textsubscript{ML}}, as given in Section 3.6.2 into SML—rules of the form:

\[
\begin{align*}
    s, A_1 \vdash \text{phrase}_1 \Rightarrow A'_1/p, s'_1 & \quad \ldots \quad s, A_n \vdash \text{phrase}_n \Rightarrow A'_n/p, s_n \\
    \text{sidecondition} & \\
    \text{map to a function of the form} \\
    \text{evalYpnosPhrase args (s, A, phrase) = let} \\
\end{align*}
\]
CHAPTER 3. IMPLEMENTATION

val A1' = evalYpnosPhrase1(s ,A1, phrase1)
...
val An' = evalYpnosPhraseN(s ,AN, phraseN)
in
  if sidecondition then A' else error("error")
end

YpnosML is a slow interpreter, as it is built on HaMLet which was not written to be fast and is not optimising. From the start of the project, I knew that extending HaMLet would make YpnosML slow, however the modular and documented HaMLet source code made extending HaMLet the best choice.

3.2.5 Compiling YpnosML

As explained in Section 4.3, the interpreted YpnosML runs slowly and is memory inefficient as it does not garbage collect grids. As YpnosML often runs many iterations of a program on large grids, having every grid generated remain in memory is infeasible.

To improve the performance and resolve memory leakage I considered the following:

Add garbage collection to HaMLet. This requires refactoring HaMLet, and a shift in the design principals. HaMLet was not designed to have internal garbage collection and many modules and data structures would need extensive modification to add this feature. HaMLet is a direct translation of The Definition of Standard ML [6] into SML, with a one-to-one mapping from a rule in the definition to a function in HaMLet. Adding garbage collection would break this link. A large requirement change can cause software to suffer from huge bugs and poor reliability.

Embed Ypnos into another implementation. Other SML compilers, such as Poly/ML, MLton and SML New Jersey are optimised for performance and have garbage collectors. However Poly/ML is the only implementation that supports multiple cores, so it is the only compiler that could be extended without adding parallelism to the compiler.

Compiling YpnosML targeting SML. HaMLet is a SML interpreter, it is not a compiler. However HaMLet has a highly modular structure so can be used to provide source-to-source compilation of SML. Ypnos can then be embedded into the SML compiler.
3.2. INTERPRETING AND COMPILING YPNOS$_{ML}$

I chose to compile Ypnos$_{ML}$ targeting SML. Poly/ML is a fast implementation of SML and supports garbage collection [7]. The Thread structure in Poly/ML has the functionality required to run an Ypnos program efficiently. This solution allowed me to leave the majority of HaMLet unaltered, as no major refactoring was required. By adding a new compiling backend and modifying the high level modules, the programmer can switch to compiling from interpreting by supplying a flag, when launching HaMLet.

Source-to-Source Compilation

I extended HaMLet to support source-to-source compilation of SML. *The Definition of Standard ML* divides the language into two sections:

**Core** is described by *The Definition of Standard ML* as a ‘lower level language’. Core contains the syntax for operations such as expressions, functions, if statements, arithmetic and logic.

**Module** is described by *The Definition of Standard ML* as ‘a middle level concerned with programming-in-the-large’. Module contains the definition of functors, structures and signatures.

My source-to-source compilation only supports the core section of SML. This supports most of the features that a programmer might wish to use in a stencil function, however it does not allow large-scale programs that use functors and structures to be compiled. Further work could include compilation of the Module system.

Source-to-source compilation is enabled by starting HaMLet with the command `./hamlet -c`. An interactive SML session is displayed to the user, however only error messages are printed to standard output. The compiled code is written to a yp-compiled.sml file. If no Ypnos extensions are used, yp-compiled.sml can be loaded by any SML implementation. If Ypnos extensions are used, then yp-compiled.sml can be loaded into Poly/ML.

The compilation mode of HaMLet uses the same lexer, parser and elaboration code as the interpreter. The addition is a new module named Comp[ilation], which performs a postorder tree traversal of the abstract syntax tree, producing SML code. The source-to-source compilation of SML is a linear representation of the abstract syntax tree. Before generating the abstract syntax tree, HaMLet applies derived forms to the code. Derived forms are mappings from SML expressions to a subset of expressions. Derived forms reduces the complexity of later compilation stages, however the mappings are heavily
influenced by λ-calculus. Instead of using if statements, logic operators and while loops, derived forms modifies the abstract syntax tree to use function closures. For example

\[ x' \text{ orelse } y' \]

is equivalent to the λ-term

\[ (\lambda fxy.fxy)\ x'\ (\lambda xy.x)\ y' \]

HaMLet uses this derived form to convert the orelse operation, during parsing:

\[ (((\text{fn } \text{true} \Rightarrow (\text{true})| \text{false} \Rightarrow (y))) (x)); \]

Derived forms cause the compiler’s target code to have many function closures, creating code that requires a large amount of garbage collection. Further work on YpnosML could reduce the number of unnecessary function closures.

**Embedding Ypnos into Compiling HaMLet**

The YpnosML compiler compiles the Ypnos extensions of YpnosML into SML code and Poly/ML threads. Compiled YpnosML takes the primitive YpnosML datatype of Grid D α and converts it into a SML α Array Array. Poly/ML arrays allow \(O(1)\) access to any element and lookups are efficient. This is important as YpnosML programs perform many lookups and the parallelisation prevents any guarantees of ordering of accessing elements.

To perform run, where a stencil function is applied in parallel to each element of a grid, the target code’s algorithm has a number of stages:

**Decompose grid pattern.** A grid pattern is converted into a list of two-tuples, positions and a list of variables, bindings. Positions represents the relative co-ordinates of a bound variable to the focal point of the grid pattern. Bindings is a list of the variables that the grid pattern binds. Both lists must be in the same order to ensure values are bound to the correct variables. For example

\[
\begin{array}{cccc}
\text{YPNOS (X*Y)} & : & a & b & c \\
\alpha\alpha & | & d & \alpha\alpha & f \\
\alpha\alpha & | & g & h & i & \Rightarrow & \phi \\
\end{array}
\]

is compiled to:
3.2. INTERPRETING AND COMPILING YPNOS$_{ML}$

\[
\text{val positions} = [(\sim 1, \sim 1), (0, \sim 1), (1, \sim 1), (\sim 1, 0), (0, 0),
(1, 0), (\sim 1, 1), (0, 1), (1, 1)]
\]

\[
\text{val bindings} = [a, b, c, d, e, f, g, h, i]
\]

Build stencil function closure. The returned list of variables is used as the parameter to a closure which computes the compiled $\phi$, $\phi_c$:

\[
\Gamma \vdash \phi \rightarrow^* \phi_c
\]

\[
\text{fn} \ [a, b, c, d, e, f, g, h, i] \Rightarrow \phi_c
\]

Convert relative co-ordinates to absolute co-ordinates. positions contains a list of co-ordinates of bound variables, relative to the focal point. When the focal point is bound to any position other than (0,0), these values are converted by adding the grid’s index to each co-ordinate. If the absolute values are outside of the grid then the co-ordinate of the nearest edge is used instead.

Lookup absolute co-ordinates in the structured grid. The Array data structure used to represent a grid allows fast access to the indices supplied. This operation is an instance of map

\[
\text{lookup} : \text{Grid} D \alpha \rightarrow (\text{int} \times \text{int}) \rightarrow \alpha
\]

\[
\text{val vs} = \text{fn lookup} =\Rightarrow \text{map (lookup)} :

((\text{int} \times \text{int}) \rightarrow \alpha) \rightarrow (\text{int} \times \text{int}) \text{ list} \rightarrow \alpha \text{ list}
\]

Apply the list of values to the stencil function. The list returned from the previous stage is used as a parameter to the expression to apply the stencil function to one focal point on the grid:

\[
(\text{fn} \ (x :: xs) \Rightarrow \phi_c) \ vs
\]

This algorithm computes one application of a stencil function to a grid, using one thread:

\[
(\text{Grid} D \alpha \rightarrow \beta) \rightarrow \text{Grid} D \alpha \rightarrow \beta
\]

Usually stencil functions are applied to each element of a grid in parallel to produce a new grid, using run. The run function performs domain decomposition on a grid to split it into a number of subgrids. run spawns worker threads that perform this algorithm.

Appendix A shows an example Ypnos$_{ML}$ program and its compiled equivalent.
3.3 Language Restrictions

Ypnos was designed to be used in a pure language, such as Haskell, where the purity guarantees that two operations performed in parallel will be mutually exclusive. SML is impure, so YpnosML is vulnerable to incorrect parallelisation of code.

Consider the program:
\[
\begin{align*}
\text{val } a &= \text{ref } 2 \\
\text{val } grid &= \text{Grid } (3 \times 2) \backslash \{ a, a, a \} \\
\text{val sten } &= \text{YPNOS } (X \times Y) : \backslash @ @ \mid @ @ x \mid @ @ = (x := !x + 1; \ 'x') \\
\text{val } grid' &= \text{run } sten \ grid
\end{align*}
\]

The result of this program cannot be guaranteed. The lack of purity in the value of \( a \) means computations on each element are not independent. When the computation is parallelised, the value of each element in the new grid is dependent on the thread scheduler. This nondeterministic behaviour is not desired and is not in line with the Ypnos ethos of transparent parallelisation.

To reduce this problem, YpnosML does not allow ref cells in grids. This includes any data structure that contains a ref cell. If a programmer attempts to build a grid with ref cells then the compiler issues a static error. As grids are \textit{statically typed}, the analysis of a grid to determine if it includes any ref cells is easy and can be incorporated into the \textit{Hindley–Milner} type inference algorithm.

YpnosML permits all other SML types, including function closures and exceptions. The only use of a ref cell is to allow state, which Ypnos is incompatible with. Other impure constructs, such as exceptions, can be stored in grids because they do not present a direct conflict with correct parallelisation and have some useful, safe purposes.

An YpnosML program with a pure grid may parallelise incorrectly when an impure stencil function is used. Consider the program
\[
\begin{align*}
\text{exception } e \\
\text{val } grid &= \text{Grid } (3 \times 1) \backslash \{ 1, 2, 3 \} \\
\text{val sten } &= \text{YPNOS } (X \times Y) : \@ @ \mid @ @ x \mid @ @ = (\text{raise } e; 3) \\
\text{val } grid' &= \text{run } sten \ grid
\end{align*}
\]

This program will be non-deterministic as any thread could throw an exception depending on the thread scheduler. Ideally, YpnosML would statically
ensure that stencil functions are pure, with programmers using monads to represent side-effects in pure SML. Unfortunately this analysis is undecidable. Consider the following stencil function where \texttt{taut(x)} is a tautology:

\begin{verbatim}
val sten = YPNOS (X*Y) : @@ | @@x | @@ = if (taut(x))
    then 3 else (raise e; 4)
\end{verbatim}

The function is guaranteed to return 3 and not raise an exception, however the \textit{halting problem} states that there is no analysis that for the general case can determine the values that a function will return. A conservative approximation could be used to determine the possible values that a function returns. Whilst this may work well for some simple stencils, the close coupling of a stencil function with a grid makes this analysis depend not just on the stencil function but on the grid:

\begin{verbatim}
exception e
val sten = YPNOS (X*Y) : @@ | @@x | @@ = if x() then 3
    else (raise e; 4)
val grid = Grid (2*1) \fn_ => taut(), fn_ => taut()
//
val grid' = Grid (2*1) \fn_ => not taut(), fn_ =>
    not taut() //
\end{verbatim}

The \texttt{sten} function is declared before the grids that it will be applied to. Therefore the type checker cannot determine when a stencil function is entered if it is a safe function as the grid that it will be applied to has not been declared. Some grids, such as \texttt{grid}, will allow this stencil function to be used, however others such as \texttt{grid'} will not. It does not make sense to allow a stencil function to be applied to some grids but not others. To do this analysis, the entire program has to be analysed in order to build a full \textit{call graph}. As YpnosML allows indirect function calls this requires \textit{alias analysis} to determine which functions could be indirectly called. This level of analysis is difficult and time consuming so is left to further work. YpnosML allows programmers to use impure stencil functions, however I encourage programmers not to use these features, as they are unsafe.

Whilst YpnosML does not guarantee safety, for most practical purposes, the language is safe. This is similar to the approach that Haskell uses with its \textit{unsafe functions}. In YpnosH, unsafe functions can be used to break parallelisation. Most practical purposes of YpnosML do not include these indirect function calls to impure functions so asking the programmer to write pure SML should not impact their code significantly.
3.4 Default Values

A stencil function binds values to variables based on their relative lexical position to the focal point. Whenever the grid pattern is larger than $1 \times 1$, the pattern will try to reference an out-of-bounds location when the focal point is near an edge. For example, the grid pattern

```
@@| a @@b c |@@
```

will reference columns to the left and right of the the edge of the grid whenever the focal point is in the leftmost or rightmost column as shown in Figure 3.4.

To avoid this problem, Ypnos recommends a number of solutions:

- The problem can be exposed to the user by binding the co-ordinates of the focal point to a variable and forcing the programmer to write a stencil function that checks each variable’s co-ordinates are not out-of-bounds before using its value. For example, the following grid pattern could bind $i$ to the co-ordinates of the focal point:

```
@@| l @@c#i r |@@
```
3.4. DEFAULT VALUES

Figure 3.5 The solid grid has been lifted to an infinite grid, represented by the dotted line. The letters $a$–$t$ represent the values at the edge of the grid, which are returned when an out-of-bounds lookup is performed.

<table>
<thead>
<tr>
<th>a</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>f</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>q</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>p</td>
<td>o</td>
<td>n</td>
<td>m</td>
<td>l</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>p</td>
<td>o</td>
<td>n</td>
<td>m</td>
<td>l</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>p</td>
<td>o</td>
<td>n</td>
<td>m</td>
<td>l</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
</tbody>
</table>

The stencil function tests the value of $i$ before referencing any variables to ensure that they will not be out-of-bounds. This requires lazy evaluation so only the variables which are dereferenced are bound. If the programmer does not correctly check $i$, an out-of-bounds exception will be thrown.

- A default value can be specified with each grid that is returned whenever an out-of-bounds lookup is performed.

- Finite grids can be lifted to infinite grids, allowing out-of-bounds accesses by specifying the behaviour of a grid when an out-of-bounds location is addressed. This can be done by reflecting the grid, extending edges to infinity or wrapping the grid so values are read from the opposite side of a grid.

By default, YpnosML lifts grids into infinite grids by extending each edge to infinity, along the extended diagonal of the grid, as shown in Figure 3.5. This is appropriate for image processing where the value of out-of-bounds lookups should be similar to the value at the nearest edge to prevent defects at the edges.

YpnosML also uses default values on a grid by using the monadic defaults
CHAPTER 3. IMPLEMENTATION

Figure 3.6 Ypnos\textsubscript{ML} code which applies \texttt{stencil} to \texttt{grid} until the sum of the element in the grid is greater than zero

\begin{verbatim}
val rec iterate = fn stencil => fn grid =>
  if reduce (fn x =>fn y =>x+y) grid > 0
    then iterate stencil (run stencil grid)
  else grid
\end{verbatim}

This is useful in structured grid computations such as the game of life where you want to precisely control values at the edge of the grid.

3.5 Making Ypnos\textsubscript{ML} Practical

The Ypnos specification includes functions to be added to the host language to allow programmers to perform common operations on grids.

3.5.1 Reductions

In scientific computing applications, it is common to apply a stencil function until a specified condition on a grid is true. For example, a programmer may want to apply a stencil function to a grid until the maximum, mean or sum of the elements of the grid meets some condition. Ypnos\textsubscript{ML} uses reductions to support this. The \texttt{reduce} function takes an \textit{associative operator} of type $\alpha \rightarrow \alpha \rightarrow (\text{Grid } D \alpha + \alpha)$ and applies the operator in parallel across a grid, by repeatedly combining two elements, as shown in Figure 3.6.

The operator should be associative to guarantee the returned value is deterministic as, due to parallelisation, Ypnos\textsubscript{ML} cannot guarantee the order in which the operator is applied. If a non-associative operator, such as subtraction, were used then different results may be returned form a call to reduce. For example, consider the $4 \times 1$ grid containing the values $3, 4, 5, 6$. The subtract operation might compute $(((3 - 4) - 5) - 6) = -12$ or $(3 - 4) - (5 - 6) = -2$. Ypnos\textsubscript{ML} does not analyse reductions to ensure they are associative—this is the programmer’s responsibility.
Reduce is similar to the higher-order fold operation on a list except that reduce does not require an initial element. fold combines the elements of a list with an initial element of type \( \beta \), however reduce combines all other elements with one of the elements of the grid, rather than an initial element.

\[
fold : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \ list \rightarrow \beta
\]

\[
reduce : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow Grid D \alpha \rightarrow \alpha
\]

3.5.2 Unzip

The Ypnos\(_{ML}\) unzip function converts a grid of tuples to a tuple of grids in parallel. This is similar to the standard unzip function that converts lists of tuples to a tuple of lists.

\[
unzip : Grid D (\alpha \ * \ \beta) \rightarrow (Grid D \alpha \ * \ Grid D \beta)
\]

With unzip Ypnos\(_{ML}\) can be used to process images that are typically specified using RGB: a three-tuple of integers in the range 0–255. This can be represented in Ypnos\(_{ML}\) by a grid of type \( Grid (X * Y) (int * (int * int)) \). The function unzip will convert this into a tuple of grids of type \( (Grid (X * Y) int * (Grid (X * Y) int * Grid (X * Y) int)) \). This allows an Ypnos\(_{ML}\) programmer to write shorter, less repetitive stencil functions.

3.5.3 Fmap

SML programmers often use the map function on lists. Map applies a function to each element of a list:

\[
map : (\alpha \rightarrow \beta) \rightarrow \alpha \ list \rightarrow \beta \ list
\]

Similarly, a function can be applied to each individual element of an Ypnos\(_{ML}\) grid, in parallel, by using the fmap function.

\[
fmap : (\alpha \rightarrow \beta) \rightarrow Grid D \alpha \rightarrow Grid D \beta
\]
3.6 The Definition of YpnosML

In this section, I provide the YpnosML grammar and semantics. HaMLet is a direct translation of The Definition of Standard ML into an interpreter with there being a mapping from semantic rules to functions. I extend the mapping to include this definition of YpnosML to functions in HaMLet. The grammar and semantics are presented in the style used by Milner [6].
3.6. THE DEFINITION OF YPNOS\textsubscript{ML}

3.6.1 Grammar

Stencil functions:

\[
\begin{align*}
\text{exp} & := \ldots \\
\text{YPNOS} & \text{ ypRule} \\
\text{YPNOSRUN} & \text{ exp DO at-} \\
& \text{exp} \\
\text{ypRule} & := \text{ypPat} \Rightarrow \text{exp} \\
\text{ypPat} & := \text{ypDimen} : \text{ypGrid} \\
\text{ypDimen} & := ( \text{ypDimenStarList} ) \\
\text{ypDimenStarList} & := \text{longVId} \\
& := \text{longVId} * \text{ypDimenStarList} \\
\text{ypGrid} & := <\text{ypRows}> \text{ ypFocalRow} <\text{ypRows}> \\
\text{ypRows} & := \text{ypRow} <\text{ypRows}> \\
\text{ypRow} & := | \text{ypTerms} | \\
\text{ypTerm} & := \text{longVId} \\
\text{ypTerms} & := \text{ypTerm} <\text{ypTerms}> \\
\text{ypFocalRow} & := @ @ | <\text{ypTerms}> @ @ \text{ypTerm} <\text{ypTerms}> @ @
\end{align*}
\]
Structured grids:
\[ \text{atexp} := \ldots \]
\[ \text{YPNOSGRID} \quad \text{ypGDimen} \quad \text{ypGD} \]
\[ \text{ypGDimen} := \langle \text{ypGDimenTerms} \rangle \quad \text{dimensions} \]
\[ \text{ypGDimenTerms} := \text{longVId} = \text{scon} \quad \text{dimension terms} \]
\[ \langle \text{ypGDimenTerms} \rangle \]
\[ \text{ypGD} := \backslash \backslash \text{ypTerms} \quad // \quad \text{row} \]
\[ \langle \text{ypGD} \rangle \]
\[ \text{ypTerms} := \text{exp} \langle \text{ypTerms} \rangle \quad \text{cell/term collection} \]

3.6.2 Semantics

Static Semantics—Type Inference

Stencil functions:

\[ C \vdash \text{ypnosRule} \Rightarrow \tau \]
\[ \overline{C \vdash \text{YPNOS} \quad \text{ypnosRule} \Rightarrow \tau} \] (3.1)
\[ C \vdash \text{ypPat} \Rightarrow \text{VE, } \tau \]
\[ \overline{C + \text{VE} \vdash \text{exp} \Rightarrow \tau'} \]
\[ \text{tynames } \text{VE} \subseteq T \text{ of } C \]
\[ \overline{C \vdash \text{ypPat} \Rightarrow \text{exp} \Rightarrow \tau \Rightarrow \tau'} \] (3.2)
\[ C \vdash \text{ypDimen} \Rightarrow D \quad C + D \vdash \text{ypGrid} \Rightarrow \text{VE, } \tau \]
\[ \overline{C \vdash \text{ypDimen : ypGrid} \Rightarrow \langle \text{VE, } \tau \rangle} \] (3.3)
\[ C \vdash \text{ypDimenStarList} \Rightarrow \tau \]
\[ \overline{C \vdash (\text{ypDimenStarList}) \Rightarrow \tau} \] (3.4)
\[ C \vdash \text{ypFocalRow} \Rightarrow \text{VE, } \tau, n \]
\[ < C \vdash \text{ypRows}_1 \Rightarrow \text{VE}', \tau, n > \]
\[ < C \vdash \text{ypRows}_2 \Rightarrow \text{VE}'', \tau, n > \]
\[ \overline{C \vdash < \text{ypRows}_1 > \text{ypFocalRow} < \text{ypRows}_2 > \Rightarrow (\text{VE} < +\text{VE}' < +\text{VE}'', \text{Grid } \tau)} \] (3.5)
\[ C \vdash \text{ypRow} \Rightarrow \text{VE, } \tau, n \quad < C \vdash \text{ypRows} \Rightarrow \text{VE}', \tau, n > \]
\[ C \vdash \text{ypRow} < \text{ypRows} > \Rightarrow (\text{VE} < +\text{VE}'', \tau, n) \]
\[ \overline{C \vdash \text{ypTerms} \Rightarrow \text{VE, } \tau, n} \]
\[ \overline{C \vdash |\text{ypTerms}| \Rightarrow \text{VE, } \tau, n} \] (3.6)
\[ \overline{C \vdash \_ \Rightarrow \{\}, \tau} \] (3.7)
\[ \overline{C \vdash \_ \Rightarrow \{\}, \tau} \] (3.8)
3.6. THE DEFINITION OF YPNOS<sub>ML</sub>

Bound Ypnos-terms type-checked by rules (34) and (35) of The Definition of Standard ML.

\[
C \vdash ypTerm \Rightarrow VE, \tau \\
< C + VE \vdash ypTerms \Rightarrow VE', \tau, n > \\
C' \vdash ypTerm < ypTerms \Rightarrow VE < +VE', \tau, n + 1
\] (3.9)

\[
C \vdash ypFocalTerm \Rightarrow VE, \tau \\
< C \vdash ypTerms_1 \Rightarrow VE', \tau, n' > \\
< C' \vdash ypTerms_2 \Rightarrow VE'', \tau, n'' > \\
C \vdash < ypTerms_1 > ypFocalTerm < ypTerms_2 > \Rightarrow VE < +VE' < +VE'', \tau, n' + n'' + 1
\] (3.10)

Structured Grids:

\[
C \vdash ypGDimen \Rightarrow \tau', D \quad C + D \vdash ypGD \Rightarrow \tau \\
C' \vdash YPNOSGRID ypGDimen ypGD \Rightarrow \tau
\] (3.11)

\[
C \vdash ypGDimen \Rightarrow \tau, D \quad C' \vdash < ypGDimen > \Rightarrow \tau, D
\] (3.12)

\[
< C \vdash ypGDimenTerms \Rightarrow \tau, D > \\
C \vdash longVID = Scon < ypGDimenTerms \Rightarrow \tau, Scon :: D
\] (3.13)

D=nil iff ypGDimenTerms = NONE

\[
C \vdash ypGTerms \Rightarrow \tau \\
< C \vdash ypGD \Rightarrow \tau > \\
C, D \vdash \backslash ypGTerms/ / < ypGD \Rightarrow \tau
\] (3.14)

\[
C \vdash exp \Rightarrow \tau \\
< C + (D - 1) \vdash ypGTerms \Rightarrow \tau > \\
C, D \vdash D = 1\oplus < ypGD >= SOME ypGD \\
C, D \vdash ypGTerms < ypGD \Rightarrow \tau
\] (3.15)

Applications:

\[
C \vdash exp \Rightarrow Grid \tau \rightarrow \tau' \\
C \vdash atexp \Rightarrow Grid \tau \\
C' \vdash YPNOSRUN \quad exp atexp \Rightarrow Grid \tau'
\] (3.16)

Dynamic Semantics—Reduction Rules

Stencil functions

\[
E \vdash YPNOS ypRule \Rightarrow (ypRule, E, \{\})
\] (3.17)
\[ E \vdash \text{exp} \Rightarrow (\text{ypRule}, E', V E) \]
\[ E \vdash \text{atexp} \Rightarrow v, D \]
\[ E + \text{Rec} V E, v, D \vdash \text{ypRule} \Rightarrow v' \]
\[ E \vdash \text{YPNOSRUN} \ \text{exp} \ \text{atexp} \Rightarrow v' \]  
(3.18)

(Decomposition of a grid into columns)

\[ m, n \vdash n > m \]
\[ E, \phi, N \vdash t = (m + 1) \ \text{div} \ N \]
\[ E, C_0 C_0 \mapsto \nu_0, C_1 C_1 \mapsto \nu_1, \ldots, C_{t-1} C_{m} \mapsto \nu_{(t(m+1)-1)} \]
\[ \text{COMPGRID} \ \text{ypPat} \Rightarrow \text{exp} \Rightarrow \nu'_1 \]
\[ E, C_t C_0 \mapsto \nu_{(t(m+1))}, C_t C_1 \mapsto \nu_{(t(m+1)+1)}, \ldots, C_{(2t-1)} C_{m} \mapsto \nu_{(2t(m+1)-1)} \]
\[ \text{COMPGRID} \ \text{ypPat} \Rightarrow \text{exp} \Rightarrow \nu'_2 \]
\[ \ldots \]
\[ E, C_t (N-1-1) C_0 \mapsto \nu_{tn(m-1)}, C_{(m-1)} C_1 \mapsto \nu_{(t(m-1)+1)}, \ldots, C_{m} C_m \mapsto V_{(n+1)(m+1)} \]
\[ \text{COMPGRID} \ \text{ypPat} \Rightarrow \text{exp} \Rightarrow \nu'_r \]
\[ E, (C_{0,0} = \alpha) \ldots (C_{m,n} = \beta), N \vdash \text{ypPat} \Rightarrow \text{exp} \Rightarrow \nu'_1 + \nu'_2 + \ldots + \nu'_r \]  
(3.19)

(Decomposition of a grid into rows)

\[ E, C_0 C_0 \mapsto \nu_0, \ldots, C_i C_j = \nu_{(i+1)(j+1)}, x \mapsto 0, y \mapsto 0 \vdash \text{ypPat} \Rightarrow V E_0 \]
\[ E, V E_0 \vdash \text{exp} \Rightarrow D_0 D_0, \]
\[ E, C_0 C_0 \mapsto \nu_0, \ldots, C_i C_j = \nu_{(i+1)(j+1)}, x \mapsto 1, y \mapsto 0 \vdash \text{ypPat} \Rightarrow V E_1 \]
\[ E, V E_1 \vdash \text{exp} \Rightarrow D_0 D_1, \]
\[ \\
\[ E, C_0 C_0 \mapsto \nu_0, \ldots, C_i C_j = \nu_{(i+1)(j+1)}, x \mapsto i, y \mapsto j \vdash \text{ypPat} \Rightarrow V E_{N-1} \]
\[ E, V E_{N-1} \vdash \text{exp} \Rightarrow D_i D_j \]
\[ E, C_0 C_0 \mapsto \nu_0, \ldots, C_i C_j \mapsto \nu_{(i+1)(j+1)} \vdash \text{COMPGRID} \ \text{ypPat} \Rightarrow \text{exp} \Rightarrow \]
\[ D_0 D_0 + D_0 D_1 + \ldots + D_i D_j \]  
(3.21)
3.6. THE DEFINITION OF YPNOS\textsubscript{ML}

\[ < E, x, y \vdash ypRows_1 \Rightarrow VE, x', y' > \]
\[ E, x' + 1, y' + 1 \vdash ypFocalRow \Rightarrow VE' \]
\[ \frac{< E, x', y' + 2 \vdash ypRows_2 \Rightarrow VE'', x'', y'' >}{E, x, y \vdash < ypRows_1 > ypFocalRow < ypRows_2 > \Rightarrow VE < +VE' > < +VE'' > } \quad (3.22) \]

\[ E \vdash ypRow \Rightarrow VE \quad < E \vdash ypRows \Rightarrow VE' > \]
\[ E \vdash ypRow < ypRows > \Rightarrow VE < +VE' > \]

\[ E \vdash ypTerm \Rightarrow VE \quad < E \vdash ypTerms \Rightarrow VE' > \]
\[ E \vdash ypTerm < ypTerms > \Rightarrow VE < +VE' > \]

\[ < E \vdash ypTerms_1 \Rightarrow VE' > \]
\[ E \vdash ypTerm \Rightarrow VE \]
\[ < E \vdash ypTerms_2 \Rightarrow VE'' > \]
\[ E \vdash \emptyset | < ypTerms_1 > \emptyset ypTerm < ypTerms_2 > | \emptyset \Rightarrow VE < +VE' > < +VE'' > \]

Structured Grids

\[ E \vdash ypGDimen \Rightarrow VE \quad E \vdash ypGD \Rightarrow VE' \]
\[ E \vdash YPNOSGRID \quad ypGDimen \quad ypGD \Rightarrow VE + VE' \]

\[ E \vdash ypGDimenTerms \Rightarrow VE \]
\[ E \vdash < ypGDimenTerms > \Rightarrow VE \]
\[ < E \vdash ypGDimenTerms \Rightarrow VE', n' > \]

\[ E, n' \vdash longVId = scon < ypGDimenTerms > \Rightarrow \{ n' \rightarrow scon \} < +VE' >, n' + 1 \]

(where \( n' = 0 \) if \( ypGDimenTerms = \text{NONE} \))

\[ E \vdash ypGTerms \Rightarrow VE \quad < E \vdash ypGD \Rightarrow VE' > \]
\[ E \vdash \backslash ypGTerms/\backslash < ypGD > \Rightarrow VE < +VE' > \]

\[ E \vdash exp \Rightarrow VE \quad < E \vdash ypGTerms \Rightarrow VE' > \]
\[ E \vdash exp < ypGTerms > \Rightarrow VE < +VE' > \]

\[ \quad (3.23) \]

\[ \quad (3.24) \]

\[ \quad (3.25) \]

\[ \quad (3.26) \]

\[ \quad (3.27) \]

\[ \quad (3.28) \]

\[ \quad (3.29) \]

\[ \quad (3.30) \]
Chapter 4

Evaluation

As a parallel programming language, YpnosML should produce fast code that shows decreasing runtime costs as more threads are used. In this section, I perform experiments on YpnosML, showing that it is 15,000 times faster than YpnosH and speedups of 50% can be gained from using more threads in an YpnosML program. My experiments show that YpnosML gives correct and repeatable results for valid inputs.

4.1 Experimental Setup

4.1.1 Evaluation Algorithms

This evaluation focuses on running programs in YpnosML. There is no existing test suite of Ypnos benchmarks. I have created YpnosML evaluation programs based on the following algorithms:

Conway’s Game of Life uses a grid that represents a two-dimensional grid of biological cells that are either alive or dead. A stencil function is applied to each cell to determine if it should be alive in the next iteration. This is computationally simple and demonstrates the ease of writing YpnosML programs. Its stencil function is:

\[
\text{val life} = \text{YPNOS} (X \ast Y) : | a \ b \ c | \ \\
| d \ e \ f | \ \\
| g \ h \ i | =>
\]

let

\[
\text{val tot} = (a+b+c+d+e+f+g+h+i)
\]
in
if (e=1) then
  if (tot<3 or else tot>4)
    then 0
  else 1
else
  if (tot=3)
    then 1
  else 0
end;

**Gaussian blur** applies a matrix convolution to pixels in an image to replace each pixel by a weighted average of its previous value and its neighbours’ values [24]. I convolve the image with a $7 \times 7$ matrix generated when the standard deviation of the weighted values, $\sigma = 0.84089642$.

$$
\begin{bmatrix}
0.00000067 & 0.00002292 & 0.00019117 & 0.00038771 & 0.00019117 & \ldots \\
0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & \ldots \\
0.00019117 & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & \ldots \\
0.00038771 & 0.01330373 & 0.11098164 & 0.22508352 & 0.11098164 & \ldots \\
0.00019117 & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & \ldots \\
\ldots
\end{bmatrix}
$$

```java
val Gaussian = YPNOS (X*Y) :
  | r0c0 r0c1 r0c2 r0c3 r0c4 r0c5 r0c6 |
  | r1c0 r1c1 r1c2 r1c3 r1c4 r1c5 r1c6 |
  | r2c0 r2c1 r2c2 r2c3 r2c4 r2c5 r2c6 |
  | @ @ r3c0 r3c1 r3c2 @ @ r3c3 r3c4 r3c5 r3c6 |
  | r4c0 r4c1 r4c2 r4c3 r4c4 r4c5 r4c6 |
  | r5c0 r5c1 r5c2 r5c3 r5c4 r5c5 r5c6 |
  | r6c0 r6c1 r6c2 r6c3 r6c4 r6c5 r6c6 |
  | => (0.00000067*r0c0)+(0.00002292*r1c1)+(0.00019117*
  | r1c1)+...
```

**Edge detection** can be applied to an image to find edges. This has medical uses in finding objects in x-rays and in image manipulation programs to help designers to select part of an image. I perform edge detection using a Sobel operator *jain1995machine* where two matrix convolutions
are applied to a grid and their results are combined:

\[
L_x = \begin{bmatrix}
-1 & 0 & 1 \\
-2 & 0 & 2 \\
-1 & 0 & 1 \\
\end{bmatrix}
\]

\[
L_y = \begin{bmatrix}
+1 & 2 & +1 \\
0 & 0 & 0 \\
-1 & -2 & -1 \\
\end{bmatrix}
\]

\[
G = \sqrt{L_x^2 + L_y^2}
\]

Its stencil function is:

\[
\text{val sobel = YPNOS (X+Y)} : | \ a \ b \ c | \\
\ | \ d \ e \ f | \\
\ | \ g \ h \ i | \\
\math{\text{Math.sqrt(Math.pow(Real.fromInt(1}*a+c-2*d+2*f-g+i),2.0) + Math.pow(Real.fromInt(a+2*b+c-g-2*h-i),2.0))}}
\]

### 4.1.2 Image Converters

To perform edge detection and Gaussian blur on photographs, I had to import photos into YpnosML. HaMLet currently does not have a graphics library. One solution to this problem would have been to build one, however this would have been a time-consuming process. Instead, YpnosML has a Java application that converts images to the syntax of YpnosML grids and can convert grids back into images.

### 4.1.3 Objectives

Throughout this evaluation, I experiment to answer four questions:

**Are YpnosML results correct?** Graphics algorithms, as well as all other algorithms can be checked by comparing the result of the program when run in YpnosML with the result when run in another application.
How does the performance of Ypnos_{ML} compare with Ypnos_{H}? By using the profiling tools in Poly/ML and the Unix `time` program, I examine the runtime performance of Ypnos_{ML} and Ypnos_{H}.

How does the performance of Ypnos_{ML} compare with C? Most performance critical programs are written in C, because it outputs fast executables. Whilst Ypnos_{ML} offers many advantages to programmers over using C, it is unlikely to be as fast. C programs are often deemed to be near-optimal so comparing the runtime performance of Ypnos_{ML} with C shows the cost of using Ypnos_{ML}.

How effective is increased Ypnos_{ML} parallelism? Ypnos_{ML} should have improving runtime performance as the number of cores used increases. By running the same computation on a varying number of cores, I can monitor the effect of increased parallelism in the computation.

4.1.4 Evaluation Hardware

This evaluation has been performed on a virtual server with a sixteen core AMD Opteron 6128 processor. Each core has a 128KB L\textsubscript{1} cache and a 512KB L\textsubscript{2} cache. There is a 12MB L\textsubscript{3} cache which is shared between all cores. The server has 1GB of RAM and is running Ubuntu 10.04 on the Xen Hypervisor.

4.2 Ypnos Correctness

When writing Ypnos_{ML} programs a programmer should be confident that the interpreter and compiler repeatedly give the correct results for their program. To guarantee Ypnos_{ML} is correct for safety-critical purposes, I would have to perform a formal proof of correctness. Given the time required to prove correctness, I leave the proof to further work.

To show programs are deterministic, I have run each program one hundred times on the same input, ensuring that the result of each computation is identical. I have also compared the result of each application with a base standard:

**Conway’s Game of Life.** I have compared the result after 100 iterations of the `glidergun` pattern with the result from `XLife`—a Game of Life implementation. Both implementations generate the same grid.
4.3. COMPARISON BETWEEN YPNOS\textsubscript{ML} AND YPNOS\textsubscript{H} 41

I have verified simple patterns by eye, for example Figure 4.1 shows the blinker pattern after one iteration of Conway’s Game of Life, as correctly calculated by Ypnos\textsubscript{ML}.

**Figure 4.1** The blinker pattern before and after a single iteration of Conway’s Game of Life

\[
\begin{array}{c|c}
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
\end{array}
\quad|\quad
\begin{array}{c|c}
0 & 0 & 0 \\
1 & 1 & 1 \\
0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{i n t } & \text{g r i d} \\
\end{array}
\quad|\quad
\begin{array}{c|c|c}
\text{i n t } & \text{g r i d} \\
\end{array}
\]

**Gaussian Blur.** Gaussian blur can be checked by eye. Figure 4.2 shows an original image and its Gaussian blur as calculated by Ypnos\textsubscript{ML}.

I have performed a Gaussian blur on Photoshop CS5, using the same filter as I have written in Ypnos\textsubscript{ML}. Both images were using the lossless PNG format, to avoid compression quantisation errors. I cropped the seven pixels closest to each edge, as Photoshop handles edge cases differently to Ypnos\textsubscript{ML}, and took checksums of both images, which were the same. This indicates that the Gaussian blur performed by Photoshop and Ypnos\textsubscript{ML} are identical, except at the edges, as anticipated.

**Edge detection.** Edge detection can be checked by comparing the output image with the original to ensure that the edges in the original are highlighted in the outputted image. Figure 4.2 shows an original image and the output of applying edge detection to that image.

Ypnos\textsubscript{ML} gives the expected result when each of these algorithms are computed and always gives the same result. This indicates that Ypnos\textsubscript{ML} is both deterministic and correct.

4.3 Comparison between Ypnos\textsubscript{ML} and Ypnos\textsubscript{H}

4.3.1 Speed

Ypnos\textsubscript{ML} is the second implementation of Ypnos. The first (proof-of-concept) implementation, Ypnos\textsubscript{H} is embedded into Haskell. The implementation is fragile—it targets multiple processor cores, however, can only use two threads and introduces “bad behaviour”. Ypnos\textsubscript{ML} can scale the number of threads it uses, providing that the grid is large enough.
Figure 4.2 Sobel operator edge detection and Gaussian blur applied to an image of dimensions $240 \times 320$px.

I have compared the time to perform structured grid computations using Ypnos$_{\text{ML}}$ and Ypnos$_{\text{H}}$. The experimental setup of Ypnos$_{\text{ML}}$ uses compiled and interpreted backends, both with two threads. Ypnos$_{\text{H}}$ has been compiled with the `-O` option for GHC, the Haskell Compiler, which produces compile time optimisations.
4.3. COMPARISON BETWEEN YPNOS\textsubscript{ML} AND YPNOS\textsubscript{H}

<table>
<thead>
<tr>
<th>Test case</th>
<th>YPNOS\textsubscript{ML}</th>
<th>YPNOS\textsubscript{H} (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Blur on an image of dimensions 240 × 80px</td>
<td>4543 (ms)</td>
<td>17</td>
</tr>
<tr>
<td>Game of Life on a grid of dimensions 100 × 100px</td>
<td>3427 (ms)</td>
<td>17</td>
</tr>
</tbody>
</table>

Compiled YPNOS\textsubscript{ML} has the best performance of all three Ypnos implementations and YPNOS\textsubscript{H} has the worst performance. Compiled YPNOS\textsubscript{ML} runs Conway’s Game of Life 15,000 times faster than YPNOS\textsubscript{H} and Gaussian blur is 830,000 times faster. This is due to inefficient communication in the YPNOS\textsubscript{H} compiler.

Compiled YPNOS\textsubscript{ML} is faster than interpreted YPNOS\textsubscript{ML} because of the optimisations gained from targeting Poly/ML. When YPNOS\textsubscript{ML} is interpreted, HaMLet has to analyse each node of the parse tree each time it is executed and then perform the instruction. However, when YPNOS\textsubscript{ML} is compiled to Poly/ML, Poly/ML compiles the SML to machine code so there is no interpretive overhead.

Compiling YPNOS\textsubscript{ML} to Poly/ML is also faster because of the optimisations built into Poly/ML [4]. Poly/ML uses inline expansion whereby the overheads of a function call are eliminated by replacing a call to a function with the function itself. YPNOS\textsubscript{ML} programs have a large number of function calls, so by removing the overheads of generating stack frames for each call, the performance is improved.

As explained in Section 3.2.5, Hamlet applies derived forms after parsing. This produces code that has many function closures. Function closures cannot use a conventional FILO call stack for stack frames as a closure can live beyond its parent and still use its parents local variables:

\[
\text{fun x a=fn ()=>a}
\]

Calling \texttt{x} will return a function closure of type \texttt{unit \rightarrow \alpha}. As the function call to \texttt{x} returns, the closure still points to a value on \texttt{x}'s stack. If the stack frame is popped, then the function closure will point to an invalid memory location. To allow functions to return closures, Poly/ML uses a spaghetti stack where frames are not stored in a FILO stack but as a graph. The Poly/ML garbage collector is used to remove frames from the spaghetti stack, however, this is expensive. Inlining function calls to reduce the work of the garbage collector improves compiled Poly/ML’s performance [4]. There are a lot of functions...
that inline expansion does not inline—this produces a large overhead that hinders performance.

### 4.3.2 Memory Footprint

Ypnos\textsubscript{ML} processes large grids, often applying a function to the grid many times until some condition is true. Repeated iteration of a stencil function over a large grid produces a large amount of data.

Figure 4.3 shows a graph of the memory use of interpreted and compiled Ypnos\textsubscript{ML}. The footprint of both backends is comparable, however HaMLet does not have an internal garbage collector, so interpreted Ypnos\textsubscript{ML} does not collect garbage. When a function is repeatedly applied to a grid, interpreted Ypnos\textsubscript{ML} keeps all previous grids in memory, which is expensive. For example, running the game of life 100 times on a $100 \times 100$ grid uses $100 \times 215 = 21,500$MB.

As most computers do not have this amount of memory available, interpreted Ypnos\textsubscript{ML}’s performance suffers as it uses all available memory and causes the operating system to page extensively. The operating system on my evaluation server sends the \textit{SIGKILL} command to HaMLet when it uses more than 1GB of memory, causing it to terminate.

This unbounded use of memory is not practical. Compiled Ypnos\textsubscript{ML} does not have memory leaks so, as well as being faster, is a more viable option for use with repeated application of a stencil function.

### 4.4 Parallelism

As a parallel language, Ypnos\textsubscript{ML} uses threads running on a chip multiprocessor to improve the runtime performance of programs. In this section, I evaluate the effect of using more cores in Ypnos\textsubscript{ML} programs. All parallel programs are limited by the amount of code that is executed sequentially. In Ypnos\textsubscript{ML}, a large amount of time is spent executing sequential code that spawns threads and combines results of computing subgrids. When larger grids are used, these overheads represent a smaller fraction of each program. I therefore perform evaluations on a variety of sizes of input to examine how the size of grid influences runtime performance.
Figure 4.3 Graph of Ypnos$_{ML}$ memory footprint against the number of rows of length 100 in a grid.
I consider the parallelisation of compiled YpnosML because it has superior performance to interpreted YpnosML. Any computations that are sufficiently expensive to require parallelisation to compute would be run using compiled YpnosML. I have performed five repeats of each experiment and plotted the the mean of their values and error bars of \( \pm \frac{\sigma^2}{n} \).

### 4.4.1 Game of Life

**Small Grid**

Figure 4.4 shows the time taken to play Conway’s Game of Life on a grid of size 100 × 100 against the number of cores used. This graph shows no significant performance gains from increased parallelism. Conway’s Game of Life is a cheap stencil function that requires few machine instructions to be processed per call. According to Poly/ML’s profiling tools, only 4% of the time is spent applying the life stencil function on a small grid. The improvements that are made from parallelising just 4% of a program are balanced out by the overheads of each thread.

**Large Grid**

Figure 4.5 shows a graph of the time taken to play Conway’s Game of Life on a grid of size 1000 × 256 against the number of cores used. Between one and six threads there is a performance increase however as more threads are added the performance worsens. This is typical instance of Amdahl’s Law: the maximum speedup is bound by the sequential part of the program. In YpnosML, this is the process of building an empty grid, spawning worker threads, returning the grid when the computation is completed and Poly/ML’s stop-the-world, sequential garbage collection. This portion of YpnosML programs does not benefit from the parallelisation and prevents runtime speeds becoming less than 1500ms. After this performance limit has been hit, increasing the parallelisation worsens performance, due to competition for cache lines. When \( n \) threads are used each thread can use \( \frac{1}{n} \) of the shared \( L_3 \) cache. \( \lim_{n \to \infty} \frac{1}{n} = 0 \) so as more threads are used each thread has a smaller portion of the \( L_3 \) cache. As the \( L_1 \) and \( L_2 \) caches, which are not shared, are only 128KB and 512KB, YpnosML relies on each thread having a lot of lines in the \( L_3 \) cache. As the number of threads increases the reduced number of lines per thread becomes costly and causes a large number of cache misses. Each cache miss stalls the program whilst a round-trip
4.4. PARALLELISM

**Figure 4.4** Graph of the time taken to compute 100 iterations of Conway’s Game of Life on a grid of size 100 × 100 against the number of cores

Graph of the Time Taken to Compute 100 iterations of Conway’s Game of Life on a 100x100 Grid Against The Number of Cores Used

The graph shows the execution time in milliseconds (ms) on the y-axis against the number of cores on the x-axis. The graph indicates that as the number of cores increases, the execution time decreases, with a plateau around the 5–10 core range, likely due to cache effects.

4.4.2 Gaussian Blur

Figure 4.6 shows the runtime performance of a Gaussian blur on images of varying sizes. For all sizes of photo, there is a clear performance gain from increased parallelism, however there are some interesting effects.

The 470×760px image sees a linear increase between one and five threads, but at five threads the graph plateaus until ten threads, due to caching. Between one and five threads the bottleneck on performance is the amount of CPU power dedicated to performing the blurring. As more cores are added this bottleneck is reduced. Between 5–10 threads there is a slight reduction in performance: as more threads are used the bottleneck is now in transferring to main memory. As the number of misses increases, the performance decreases.
data from memory into caches and from caches into the processor, which is not improved by having more threads. There is another speedup when 10–16 threads are used. As more threads are used, subgrids become smaller. When ten threads are used most of a subgrid can fit into the $L_1$ and $L_2$ caches, so the cores do not need to use the $L_3$ cache, improving performance. If more cores were available I hypothesise that it would plateau for a number of threads and then marginally improve again as subgrids begin to fit onto the $L_1$ cache. I hypothesise that this would occur when 50 threads are used, as the $L_1$ cache is $\frac{1}{5}$ of the size of the combined $L_1$ and $L_2$ caches.

The $240 \times 320$ px line shows initial improvements, as the number of threads increases from one to three. Beyond three threads there appears to be an improvement in runtime performance, however, the error bars are too large to tell if the same effect occurs with subgrids fitting into the $L_1$ and $L_2$ caches.

Figure 4.6 has large error bars. The large grid pattern used by Gaussian
blurring causes a large amount of data to be moved between the main memory and caches. Whenever data is fetched from main memory, a cache line is evicted from the set of locations generated from the index bits of the requested address. On the Opteron 6128 processor, the caches are sixteen-way set associative, so each address can be placed into one of sixteen different cache lines. If a computation has a large number of lines evicted that are subsequently addressed by another thread, then there are lots of cache misses so the program runs slowly. However, if addresses evict lines that are not later requested, then there are fewer cache misses so the program performs faster. The non-determinism of parallel applications causes threads to be executed at different speeds on each program run, causing some runs to suffer from this problem significantly, whereas other do not. Gaussian blurring has a larger grid pattern than the other programs, so is more data intensive, which increases the error margins.

The error bar for one thread is small because there are no other threads com-
peting for cache lines. The thread accesses a small range of memory locations and the processor is designed not to map nearby addresses to the same cache lines. For two threads, the error bars are the largest of any number of threads because the two threads compete for cache lines and when the caches miss, one of the threads has to sleep, so the program only uses 50% of the available processing power. As more threads are added, the cost of having two threads that compete becomes less as the threads perform a smaller fraction of the overall program. The bandwidth between the processor and main memory becomes a bottleneck, so lookups are buffered, causing the latency of lookups to increase. Whilst increased latency is generally bad, it reduces the issue of cache conflicts because if two threads compete, when one performs a lookup, the increased lookup time allows the thread that it is competing with to make more progress, so the threads become out of sync and should stop competing. These effects cause the error bars to shrink when more than two cores are used.

The errors are larger when an even number of threads, \( n \), are used. This is also due to threads competing for cache lines. When an even number of threads are used there is more cache aliasing, where multiple addresses map to the same set of cache lines. The processor uses the bits 5–8 as the index bits for the cache. If the grid contains \( n \) addresses and \( m \) threads are used each thread starts \( \frac{n}{m} \) addresses apart. If \( 2r = m \) with \( r \in \mathbb{Z} \), then the difference in addresses is equivalent to calculating \( \frac{n}{m} \) and then performing a bit-shift to the right, so index bits now contain a bit that was outside the index bits. Using higher bits as an index increases cache aliasing because they rarely change. When \( 2^3 = 8 \) threads are used, 3 shift-rights occur to the address bit pattern. The bits which were 8–11 are now the index bits. These bits are poorly distributed the caches alias.

### 4.4.3 Edge Detection

Figure 4.7 shows a graph of the time taken to detect edges on an image against the number of cores used [24]. The algorithm is performed on the same images as used in Figure 4.6. I omit analysis of edge detection as the analysis applied to Figure 4.6 applies to Figure 4.7.
4.5. Comparison with Optimised C

Many performance critical programs are written in C. The language is low level, allowing code that targets fast implementation, to which many safe optimisations can be applied at compile time, so runtime speeds are fast.

Appendix B shows a structured grid computation, written in C. The program runs a Gaussian blur on an image of dimensions $480 \times 760$px, using a matrix convolution. The code has been optimised to run quickly using 16 threads and the convolution is applied using the same algorithm as used in YpnosML.

The C program runs in 8.4 seconds, whereas YpnosML runs the same program in 16.8 seconds. The C program is clearly faster than its YpnosML equivalent, however this difference can be partially bridged by hardware performances.

The time taken to write and debug the C program is far higher than the time taken to write an YpnosML program. When using C, the programmer has to use pointers, spawn threads, consider edge cases, perform printing...
The algorithm requires 150 lines of code. This contrasts the YpnosML approach, where using a functional host language with embedded Ypnos features, lets the programmer write two lines of code to perform all these tasks.

### 4.6 Future Performance Improvements

The performance of interpreted YpnosML is quite slow, however this is due to inefficiencies in HaMLet. If programmers want their code to run quickly then they should use the compiled backend which offers far greater performance. In this section, I focus on ways that future work can improve the runtime performance of compiled YpnosML.

**Grid pattern based domain decomposition.** Often YpnosML has stencil functions that allow domain decomposition to be performed in a more efficient way. For example, the stencil function

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

only accesses one row of a grid. If domain decomposition is performed by building subgrids that are always rows of the original grid, then computing a subgrid in iteration \(n+1\) only depends on the same subgrid in iteration \(n\)—there are no dependencies on other subgrids. Concurrency controls could be weakened as there is no need to finish computing all subgrids in iteration \(n\) before beginning to compute iteration \(n+1\).

In YpnosML, threads block until the entire previous iteration is complete before beginning the next iteration to avoid concurrency issues at the edge of subgrids. The grid patterns could be used to process the region that other threads will use in iteration \(n+1\) with a higher priority so that if the other thread finishes the current iteration first, it can use the values already calculated to begin the next iteration. Similarly, rather than blocking, each thread could compute the inside of the next iteration, for elements which when made the focal point do not reference elements outside of the subgrid.

**Better cache management.** YpnosML does not modify its domain decomposition based on the underlying hardware. From the evaluation results, it is clear that YpnosML is limited by its use of caches and under some conditions its performance suffers from a high level of cache misses. Improved cache blocking could be used, whereby each thread
processes subgrids by splitting them into sub-subgrids, that fit into the L1 cache and processing the entire sub-subgrid, before moving onto the next sub-subgrid.

**Reduced sequential overheads.** As with all parallel applications, YpnosML is limited in its performance gains by the sequential part of the programs. In YpnosML programs, the largest sequential part of code is the garbage collector. Poly/ML uses a sequential, stop-the-world garbage collector that runs for about $1/5$ of the time that an YpnosML program executes. If the Poly/ML garbage collector were rewritten to be a parallel, concurrent garbage collector, then the overheads would be reduced. YpnosML would not need to be modified.

**Fewer closures.** When stencil functions are compiled to SML, they contain a large number of function closures, due to SML derived forms. These function closures create more objects on the heap that the garbage collector has to traverse and are partially responsible for the large amount of time spent performing garbage collection. I have hand-modified some YpnosML target code to reduce the number of function closures and have found that this substantially decreases the time the garbage collector spends running. Performance could be increased by modifying the compiler to use fewer function closures.

### 4.7 Overview

Throughout this evaluation I have shown that interpreted and compiled YpnosML met my initial objectives and two of the extensions given in Appendix C.8.9. The remaining evaluation, to support many-dimensional patterns, has been left to further work due to syntactical difficulties.

I have tested my interpreter and compiler against a variety of test programs, showing that they repeatedly give the expected results for the input. I wrote a program in C that is equivalent to an YpnosML program—whilst the C program was significantly faster, it was highly optimised. The C program required more programmer effort than the YpnosML program because it does not have the same abstractions and syntactical convenience of YpnosML.

Both backends of YpnosML have been shown to be substantially faster than YpnosH, making YpnosML the fastest existing implementation of Ypnos. The parallelisation in YpnosML is scalable to the number of available processor cores and uses more sophisticated parallelisation than YpnosH. The YpnosML
parallelisation improves the runtime performance of YpnosML when large grids are used. The effect of parallelisation is far smaller on smaller grids, as expected. The performance increases are bound by Amdahl’s Law and the limits of caches in the processor.
Chapter 5

Conclusions

This project has been a success. I have successfully built a fast and correct embedded domain specific language that can be compiled or interpreted. Both backends run in parallel, on a scalable number of cores. YpnosML produces repeatable, correct results. It has been used to create parallel structured grid programs, as shown in Chapter 4.

I have met all of my initial criteria. YpnosML lets programmers specify two dimensional data structures as grids. These grids can be of any size. To improve correctness of parallelisation, YpnosML grids prohibit ref cells from being used in its elements. Ypnos stencil functions can be written using any SML expression combined with a grid pattern, mapping values to variables. Using the \texttt{run} function YpnosML applies a stencil function to each element of the grid in parallel. Section 4 shows that the result from using \texttt{run} is correct and gives a detailed performance analysis of YpnosML.

YpnosML includes two extensions from my proposal—type inference and reductions. Type inference is used in the elaboration module of HaMLet to detect the element type stored in a grid, using the Hindley–Milner type inference algorithm. Reductions apply an associative operator across a grid to reduce a grid to a single value. I have implemented extensions that were not initially proposed: \texttt{zip}, \texttt{fmap} and compiling YpnosML. Compiling YpnosML was a major extension that required modifying HaMLet, an ML interpreter to add a compiling backend. This extension improved YpnosML performance.

YpnosML integrates with SML, by extending the SML syntax in the same style and adding grids as a primitive ML type. The right-hand side of a stencil function uses the SML \texttt{exp} non-terminal. This is an improvement
over YpnosH where quasi-quoting reduces the available grammar for stencil functions.

5.1 Performance

My evaluation of YpnosML shows that both interpreted and compiled YpnosML are the fastest existing implementations of Ypnos. They are substantially faster than YpnosH. Whilst I expected interpreted YpnosML to be quite slow, its main problem is the lack of garbage collection. Whenever large grids are used, interpreted YpnosML uses excessive amounts of memory, causing the operating system to kill YpnosML. This fault lies with HaMLet, rather than YpnosML, in that HaMLet was not designed to handle large amounts of data.

I successfully overcame the problems with interpreted YpnosML by building the YpnosML compiler that targets Poly/ML. By compiling Poly/ML there is a substantial increase in runtime performance. Poly/ML’s garbage collection and code optimisations produce programs that run faster than interpreted YpnosML and do not suffer from memory leaks.

Compiled YpnosML is slower than programs written in C. The C program that I wrote was hand optimised to be fast and the GCC compiler is very advanced with better optimisations than Poly/ML. The gap could be closed by adding optimisations to the YpnosML compiler or improvements in hardware. The ease and speed of writing YpnosML programs, compared with C programs, makes YpnosML a viable option for running scientific computations.

YpnosML and YpnosH both exploit the data level parallelism in structured grid computations by performing domain decomposition on a chip multiprocessor. Chip multiprocessors are not specialised for applying the same instructions to multiple data, however GPUs are designed for these single instruction multiple data operations. GPUs rapidly apply operations to large matrices. An implementation of YpnosML that would give far greater performance, would be to target CUDA, to exploit the SIMD property of GPUs.

5.2 Improvements

YpnosML avoids the problems caused by enforcing the SIW property in an impure language by asking the programmer to write sensible code and limiting
the use of ref cells in grids. Whilst this is in the style of Ypnos, as all Ypnos implementations require the programmer to write associative reduction functions, it is not the most elegant solution. YpnosML could be extended to perform whole-program analysis to verify that stencil functions are pure. Alternatively, I could investigate the benefits of impure stencil functions.

YpnosML allows many dimensional grids to be specified by building grids of grids. However, it does not allow many dimensional stencil functions to be specified without nesting stencil functions. The Ypnos specification does not give any examples of syntax that can be used to draw three-dimensional grid patterns, so this would need to be decided upon.

5.3 Overview

This project has been a success: I have met all the proposed success criteria and implemented two proposed extensions. I further extended the project to build the backend of a SML compiler and embed Ypnos into this compiler. The YpnosML compiler and interpreter are the fastest existing Ypnos implementations and the only implementations that can scale the number of cores that they run on. The results given by YpnosML have been shown to be repeatedly correct.

Whilst using YpnosML does not produce code that is faster than writing a program in C, the time spent writing the program is reduced. The features in YpnosML such as functional programming, compiler-drive parallelisation, automatic out-of-bounds handling and lack of annotation make YpnosML a useful language for a functional programmer.
Appendix A

Compiled Ypnos Gaussian Blur

val toReal = (fn ([a]) =>
  let
    val {1 = x, 2 = y, 3 = z} = a;
  in
    {1 = (Real.fromInt) (x), 2 = (Real.fromInt) (y),
     3 = (Real.fromInt) (z)}
  end, (0,0)::[]);
val tupMap = (fn x =>
  let
    val {1 = m, 2 = {1 = r, 2 = g, 3 = b}, 3 = {1 = x, 2 = y, 3 = z}} = x;
    fun tupMap (r,g,b)=(r,g,b)
  in
    {1 = ( + ) ({1 = (Real.round) (( *) ({1 = m, 2 = r})), 2 = x}), 2 = ( + ) ({1 = (Real.round) (( *) ({1 = m, 2 = g})), 2 = y}), 3 = ( + ) ({1 = (Real.round) (( *) ({1 = m, 2 = b})), 2 = z})
  end
 ));
val gaussian = (fn ([r0c0, r0c1, r0c2, r0c3, r0c4, r0c5, r0c6, r1c0, r1c1, r1c2, r1c3, r1c4, r1c5, r1c6, r2c0, r2c1, r2c2, r2c3, r2c4, r2c5, r2c6, r3c0, r3c1, r3c2, r3c3, r3c4, r3c5, r3c6, r4c0, r4c1, r4c2, r4c3, r4c4, r4c5, r4c6, r5c0, r5c1, r5c2, r5c3, r5c4, r5c5, r5c6, r6c0, r6c1, r6c2, r6c3, r6c4, r6c5...
APPENDIX A. COMPILED YPNOS GAUSSIAN BLUR

\[
(r_{6c6}) \Rightarrow (\text{tupMap}) (\{1 = 0.00000067, 2 = r0c0, 3 = (\text{tupMap}) (\{1 = 0.00002292, 2 = r0c1, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r0c2, 3 = (\text{tupMap}) (\{1 = 0.00038771, 2 = r0c3, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r0c4, 3 = (\text{tupMap}) (\{1 = 0.00002292, 2 = r0c5, 3 = (\text{tupMap}) (\{1 = 0.00000067, 2 = r0c6, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r1c0, 3 = (\text{tupMap}) (\{1 = 0.00078633, 2 = r1c1, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r1c2, 3 = (\text{tupMap}) (\{1 = 0.01330373, 2 = r1c3, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r1c4, 3 = (\text{tupMap}) (\{1 = 0.00078633, 2 = r1c5, 3 = (\text{tupMap}) (\{1 = 0.00002292, 2 = r1c6, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r2c0, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r2c1, 3 = (\text{tupMap}) (\{1 = 0.05472157, 2 = r2c2, 3 = (\text{tupMap}) (\{1 = 0.11098164, 2 = r2c3, 3 = (\text{tupMap}) (\{1 = 0.05472157, 2 = r2c4, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r2c5, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r2c6, 3 = (\text{tupMap}) (\{1 = 0.00038771, 2 = r3c0, 3 = (\text{tupMap}) (\{1 = 0.01330373, 2 = r3c1, 3 = (\text{tupMap}) (\{1 = 0.11098164, 2 = r3c2, 3 = (\text{tupMap}) (\{1 = 0.22508352, 2 = r3c3, 3 = (\text{tupMap}) (\{1 = 0.11098164, 2 = r3c4, 3 = (\text{tupMap}) (\{1 = 0.01330373, 2 = r3c5, 3 = (\text{tupMap}) (\{1 = 0.00038771, 2 = r3c6, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r4c0, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r4c1, 3 = (\text{tupMap}) (\{1 = 0.05472157, 2 = r4c2, 3 = (\text{tupMap}) (\{1 = 0.11098164, 2 = r4c3, 3 = (\text{tupMap}) (\{1 = 0.05472157, 2 = r4c4, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r4c5, 3 = (\text{tupMap}) (\{1 = 0.00019117, 2 = r4c6, 3 = (\text{tupMap}) (\{1 = 0.00002292, 2 = r5c0, 3 = (\text{tupMap}) (\{1 = 0.00078633, 2 = r5c1, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r5c2, 3 = (\text{tupMap}) (\{1 = 0.01330373, 2 = r5c3, 3 = (\text{tupMap}) (\{1 = 0.00655965, 2 = r5c4, 3 = (\text{tupMap}) (\{1 = 0.00078633, 2 = r5c5, 3 = (\text{tupMap}) (\{1 =
61

0.00002292, 2 = r5c6 , 3 = ( tupMap) {{1 =
0.00000067, 2 = r6c0 , 3 = ( tupMap) {{1 =
0.00002292, 2 = r6c1 , 3 = ( tupMap) {{1 =
0.00019117, 2 = r6c2 , 3 = ( tupMap) {{1 =
0.00038771, 2 = r6c3 , 3 = ( tupMap) {{1 =
0.00019117, 2 = r6c4 , 3 = ( tupMap) {{1 =
0.00002292, 2 = r6c5 , 3 = ( tupMap) {{1 =
0.00000067, 2 = r6c6 , 3 = {1 = 0, 2 = 0, 3 = 0}}}})

):(:~3,~2)::(~3,~1)::(~3,0)::(~3,1)::(~3,2)::(~3,3)
::(~2,~3)::(~2,~2)::(~2,~1)::(~2,0)::(~2,1)::(~2,2)
::(~2,3)::(~1,~3)::(~1,~2)::(~1,~1)::(~1,0)::(~1,1)
::(~1,2)::(~1,3)::(0,~3)::(0,~2)::(0,~1)::(0,0)
::(0,1)::(0,2)::(0,3)::(1,~3)::(1,2)::(1,~1)::(1,0)
::(1,1)::(1,2)::(1,3)::(2,~3)::(2,~2)::(2,~1)::(2,0)
::(2,1)::(2,2)::(2,3)::(3,~3)::(3,~2)::(3,~1)::(3,0)
::(3,1)::(3,2)::(3,3)::[];

val rec rep = (fn 0=> (photo)
| n=> (let
val r =
let
val c = ref (8)
val dd = ref 0
fun foo sten grid t =
let
val m = Thread.Mutex.mutex()
val cv = Thread.ConditionVar.conditionVar()
val _ = Thread.Mutex.lock(m)
val g = fn (stenc, rels) => fn grid => fn i
=>
let
val abs = fn(j) => map (fn(i', j') => (Int.max(Int.min(i+i', Array.length(grid)) 1),0), Int.max(Int.min(j+j', Array.length(Array.sub(grid,0)) 1),0))) rels
in
Array.tabulate(Array.length(Array.sub(grid,0)), x => (stenc(map(fn (}}}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}})){}}}){}}})){}}})){}}})){}}})){}}})){}}}){}})

::(~3,~2)::(~3,~1)::(~3,0)::(~3,1)::(~3,2)::(~3,3)
::(~2,~3)::(~2,~2)::(~2,~1)::(~2,0)::(~2,1)::(~2,2)
::(~2,3)::(~1,~3)::(~1,~2)::(~1,~1)::(~1,0)::(~1,1)
::(~1,2)::(~1,3)::(0,~3)::(0,~2)::(0,~1)::(0,0)
::(0,1)::(0,2)::(0,3)::(1,~3)::(1,2)::(1,~1)::(1,0)
::(1,1)::(1,2)::(1,3)::(2,~3)::(2,~2)::(2,~1)::(2,0)
::(2,1)::(2,2)::(2,3)::(3,~3)::(3,~2)::(3,~1)::(3,0)
::(3,1)::(3,2)::(3,3)::[]}];

::(3,1)::(3,2)::(3,3)::[]);
i, j) \Rightarrow \text{Array.sub(Array.sub(grid, i), j))(abs(x))}

end
val first = Array.sub(grid, 0, 0)
val newg = Array.array(Array.length(grid),
Array.fromList([first]))
fun doGrid sten grid t n =
  let
    fun doRows f ar start n =
      | doRows f ar start n =
        (Array.update(ar, start+n, (f (start+n)))
         handle Subscript => print("error");
        doRows f ar
        start (n + 1))
    val low = (if (!dd = !c 1) then 0 else
      Int.max(0, n (Array.length(grid) div t)))
    val numRows = n + low
    in
    dd := (!dd + 1);
    Thread.Thread.fork(fn _ =>
      doRows (grid, newg low numRows;
    Thread.Mutex.lock(m);
    c := (!c 1);
    if (!c=0) then (Thread.Mutex.unlock m;
      Thread.ConditionVar.signal(cv)) else
      Thread.Mutex.unlock(m),[]);
    if (low > 0) then (doGrid sten grid t low) else
      (while (!c > 0) do (
        Thread.ConditionVar.wait(cv, m);
        Thread.Mutex.unlock(m))
    end
val d = fn sten \Rightarrow fn grid \Rightarrow fn t \Rightarrow
doGrid sten grid t (Array.length(grid) + 1)
in
(d sten grid t; newg)
end
in
  foo toReal(( rep) (( ) ({1 = n, 2 = 1}))) (!
c
end;

in
let
  val c = ref (2)
  val dd = ref 0
  fun foo sten grid t =
    let
      val m = Thread.Mutex.mutex()
      val cv = Thread.ConditionVar.conditionVar()
      val _ = Thread.Mutex.lock(m)
      val g = fn (stenc, rels) => fn grid => fn i =>
        let
          val abs = fn (j) => map (fn (i', j') => (Int.max(Int.min(i+i', Array.length(grid)), 1), 0), Int.max(Int.min(j+j', Array.length(Array.sub(grid, 0))), 0)))) rels
        in
          Array.tabulate(Array.length(Array.sub(grid, 0)), fn x => (stenc(map (fn (i, j) => Array.sub(Array.sub(grid, i), j))(abs(x))))))
        end
      val first = Array.sub(g sten (Array.fromList([Array.sub(Array.sub(grid, 0), 0)]))) 0, 0
      val newg = Array.array(Array.length(grid), Array.fromList([first])
    in
      doGrid sten grid t n =
        let
          fun doRows f ar start ~1 = ()
            | doRows f ar start n = (print("start=
            ^ Int.toString(start) ^ " finish="
            ^ Int.toString(n)^ ". ")); Array.
            update(ar, start+n, (f (start+n)))
            handle Subscript => print("error");
            doRows f ar
            start (n 1))
val low = (if (!dd = !c 1) then 0 else
Int.max(0, n (Array.length(grid) div t )))
val numRows = n low
in dd := (!dd + 1);
Thread.Thread.fork(fn => (doRows(g sten grid) newg low numRows;
Thread.Mutex.lock(m); c := (!c 1);
if (!c=0) then (Thread.Mutex.unlock(m);
Thread.ConditionVar.signal(cv))
else Thread.Mutex.unlock(m),[]);
if (low > 0) then (doGrid sten grid t low) else (while (!c > 0) do (Thread.ConditionVar.wait(cv,m));
Thread.Mutex.unlock(m)
)
)
val d = fn sten => fn grid => fn t => doGrid
sten grid t (Array.length(grid) 1)
in (d sten grid t;
newg)
in
foo gaussian( r) (!c)
end
end ));
val res = fn () => ((rep) (3));
Appendix B

Gaussian Blur Written in C

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#define NUM_THREADS 16

struct pixel {
    short unsigned int r, g, b;
};

struct pixel img[639*480], newimg[639*480], s;
int i = 0;
int pos = 0;
int height = 639;
int width = 480;

clock_t start, finish;

int parseInt(const char s[]) {
    while (!isdigit(s[+pos]));
}
pos ;
return atoi(&s[pos]) ;
}

void parseLine(const char line[]) {
  // until EOL
  int x;
  for (x=0;x<3;x++) {
    while(!(line[pos++]=')'||(line[pos]=='\0')||(line[pos]=='/') ) ;
    if (line[pos1]==10||line[pos1]=='/') {
      printf("DONE,%d",pos);
      pos = 0;
      return;
    }
    img[i].r=parseInt(line);
    while (isdigit(line[pos++]));
    img[i].g=parseInt(line);
    while (isdigit(line[pos++]));
    img[i++].b=parseInt(line);
    while (isdigit(line[pos++]));
    pos ;
  }
  pos = 0;
}

int conv(int i, int j) {
  return j * width + i;
}

struct pixel * get(int i, int j) {
  int x = i<0?0:i>width?width:i;
  int y = j<0?0:j>width?width:j;
  return &img[conv(x,y)];
}

void gaussian(void * arg) {
  int args[2] = {1,100};
  /* int * args = *((int*) arg); */
int start = args[0];
int end = args[1];
int i, j;
for (j = start; j < end; j++) {
    for (i = 0; i < width; i++) {
        newimg[conv(i, j)].r = (int)(((*get(i, j)).r * 0.00000067) +
                                   ((*get(i+1, j)).r * 0.00002292) +
                                   ((*get(i+2, j)).r * 0.00000067) +
                                   ((*get(i+3, j)).r * 0.000019117) +
                                   ((*get(i+4, j)).r * 0.000019117) +
                                   ((*get(i+5, j)).r * 0.00000067) +
                                   ((*get(i+6, j)).r * 0.000019117) +
                                   ((*get(i+7, j)).r * 0.000019117) +
                                   ((*get(i+8, j)).r * 0.00000067) +
                                   ((*get(i+9, j)).r * 0.000019117) +
                                   ((*get(i+10, j)).r * 0.000019117) +
                                   ((*get(i+11, j)).r * 0.00000067) +
                                   ((*get(i+12, j)).r * 0.000019117) +
                                   ((*get(i+13, j)).r * 0.000019117) +
                                   ((*get(i+14, j)).r * 0.00000067) +
                                   ((*get(i+15, j)).r * 0.000019117) +
                                   ((*get(i+16, j)).r * 0.000019117) +
                                   ((*get(i+17, j)).r * 0.00000067) +
                                   ((*get(i+18, j)).r * 0.000019117) +
                                   ((*get(i+19, j)).r * 0.000019117) +
                                   ((*get(i+20, j)).r * 0.00000067) +
                                   ((*get(i+21, j)).r * 0.000019117) +
                                   ((*get(i+22, j)).r * 0.000019117) +
                                   ((*get(i+23, j)).r * 0.00000067) +
                                   ((*get(i+24, j)).r * 0.000019117) +
                                   ((*get(i+25, j)).r * 0.000019117) +
                                   ((*get(i+26, j)).r * 0.00000067) +
                                   ((*get(i+27, j)).r * 0.000019117) +
                                   ((*get(i+28, j)).r * 0.000019117) +
                                   ((*get(i+29, j)).r * 0.00000067) +
                                   ((*get(i+30, j)).r * 0.000019117)) * 0.001330373;
* 0.00000067) +((
    get(i+2,j+3)).r 0.00002292) +((
    get(i+3,j+3)).r 0.00000067));
newimg[conv(i,j)].g = (int)((((
    get(i 3 , j 3)).g 0.00000
    0.00002292) +((
    get(i 2 , j 3)).g 0.00019117) +((
    get(i 1 , j 3)).g 0.00038771) +((
    get(i 0 , j 3)).g 0.00019117) +((
    get(i+1,j 3)).g 0.00002292) +((
    get(i+2,j 3)).g 0.00000067)) +((
    get(i+3,j 3)).g 0.00000067) +((
    get(i+4,j 3)).g 0.00000067) +((
    get(i 3 , j 2)).g 0.00000067) +((
    get(i 2 , j 2)).g 0.00000067) +((
    get(i 1 , j 2)).g 0.00000067) +((
    get(i 0 , j 2)).g 0.00000067);
\[
\begin{align*}
&*\text{get}(i+3,j+2).g * 0.00002292 + ((*\text{get}(i 3,j+3)).g \\
&* 0.00000067) + ((*\text{get}(i 2,j+3)).g * 0.00002292 + ((*\text{get}(i 1,j+3)).g \\
&* 0.000019117) + ((*\text{get}(i,j+3)).g * 0.00038771) + ((*\text{get}(i+1,j+3)).g * 0.00019117 \\
& + ((*\text{get}(i+2,j+3)).g * 0.00002292) + ((*\text{get}(i+3,j+3)).g * 0.00000067)) ;
\end{align*}
\]

\[
\text{newimg}[\text{conv}(i,j)].b = (\text{int})(((*\text{get}(i 3,j 3)).b * 0.00000067) + ((*\text{get}(i 2,j 3)).b * 0.00002292) + ((*\text{get}(i 1,j 3)).b * 0.00019117) + ((*\text{get}(i,j 3)).b * 0.00038771) + ((*\text{get}(i+1,j 3)).b * 0.00019117) + ((*\text{get}(i+2,j 3)).b * 0.00002292) + ((*\text{get}(i+3,j 3)).b * 0.00000067) + ((*\text{get}(i 2,j 2)).b * 0.000078633) + ((*\text{get}(i 1,j 2)).b * 0.00655965) + ((*\text{get}(i,j 2)).b * 0.00002292) + ((*\text{get}(i+1,j 2)).b * 0.00078633) + ((*\text{get}(i+2,j 2)).b * 0.00000067) + ((*\text{get}(i+3,j 2)).b * 0.00002292) + ((*\text{get}(i 3,j 1)).b * 0.00019117) + ((*\text{get}(i 2,j 1)).b * 0.00655965) + ((*\text{get}(i 1,j 1)).b * 0.05472157) + ((*\text{get}(i,j 1)).b * 0.11098164) + ((*\text{get}(i+1,j 1)).b * 0.05472157) + ((*\text{get}(i+2,j 1)).b * 0.00655965) + ((*\text{get}(i+3,j 1)).b * 0.00019117) + ((*\text{get}(i 3,j)).b * 0.00038771) + ((*\text{get}(i 2,j)).b * 0.01330373) + ((*\text{get}(i 1,j)).b * 0.011098164) + ((*\text{get}(i,j)).b * 0.000038771) + ((*\text{get}(i+1,j)).b * 0.01330373) + ((*\text{get}(i+2,j)).b * 0.000387771) + ((*\text{get}(i+3,j)).b * 0.00002292) + ((*\text{get}(i 3,j+1)).b * 0.01330373) + ((*\text{get}(i 2,j+1)).b * 0.0000655965) + ((*\text{get}(i 1,j+1)).b * 0.00000067) + ((*\text{get}(i,j+1)).b * 0.05472157) + ((*\text{get}(i+1,j+1)).b * 0.11098164) + ((*\text{get}(i+2,j+1)).b * 0.05472157) + ((*\text{get}(i+3,j+1)).b * 0.00019117) + ((*\text{get}(i 3,j+2)).b * 0.00002292) + ((*\text{get}(i 2,j+2)).b * 0.000078633) + ((*\text{get}(i 1,j+2)).b * 0.00655965) + ((*\text{get}(i,j+2)).b * 0.00000067) + ((*\text{get}(i+1,j+2)).b * 0.00002292) + ((*\text{get}(i+2,j+2)).b * 0.00078633) + ((*\text{get}(i+3,j+2)).b * 0.00000067) + ((*\text{get}(i 3,j+3)).b * 0.00002292) + ((*\text{get}(i 2,j+3)).b * 0.00000067) + ((*\text{get}(i 1,j+3)).b * 0.000019117) + ((*\text{get}(i,j+3)).b * 0.00038771) + ((*\text{get}(i+1,j+3)).b * 0.00019117) + ((*\text{get}(i+2,j+3)).b * 0.00002292) + ((*\text{get}(i+3,j+3)).b * 0.00000067).
\]
get(i+2,j+2).b * 0.00078633) +(
* get(i+3,j+2).b * 0.00002292) +((get(i3,j+3)).b
* 0.00019117) +((get(i+1,j+3)).b
* 0.00038771) +((get(i+2,j+3)).b * 0.0000229
2) +((get(i+3,j+3)).b * 0.00000067));
}
}

int main(int argc, char *argv[]) {
  char line[80];
  int f;
  int i, j;
  int rc;

  pthread_t threads[NUM_THREADS];
  int thread_args[2][NUM_THREADS] =
    {{0,320},{320,639}};

  for (i=0; i < NUM_THREADS; i++) {
    thread_args[0][i]=i*height/NUM_THREADS;
    thread_args[1][i]=(i+1)*height/
    NUM_THREADS;
    if (thread_args[1][i] > height)
      thread_args[1][i] = height;
  }

  static const char filename[] = "480x760.txt";
  FILE *file = fopen ( filename, "r" );
  if ( file != NULL )
    {
      char line [ 128 ]; /* or other suitable
maxum line size */
while ( fgets ( line, sizeof line, file ) != NULL ) /* read a line */
{
parseLine(line);
}
fclose ( file );
start = clock();
//gaussian(0, height);

for ( i=0; i<NUM_THREADS; ++i ) {
rc = pthread_create(&threads[i], NULL, gaussian, (void *)
) &thread_args[i]);
}

for ( i=0; i<NUM_THREADS; ++i ) {
rc = pthread_join(threads[i], NULL);
//assert(0 == rc);
}

for ( j=0;j<height;j++) {
for ( i=0; i<width; i++) {
 s = *get(width 10,
height 10);
 printf("%d,%d,%d),\n",(* get(i,j)).r ,(*get(i,
 j)).g
 ,(*get(i,j)).b);
}
}
finish = clock();
f = ((finish - start));
printf("\n\n\n\n",( f ));
}
else
{
printf("%s","Gaussianed");
return 0;
}
Appendix C

Project Proposal

C.1 Introduction and Description of the Work

Ypnos is a pure declarative embedded domain-specific language for structured grid computing [1]. Ypnos aims to produce highly parallelised code whilst avoiding the pitfalls of the majority of existing parallel languages: those with a compiler-driven approach to parallelisation which produce unpredictable code and those which use annotated code to perform parallelisation, but are difficult for programmers to use. Ypnos aims to solve this problem for structured grid computations by enforcing the property of single, independent writes which enables the compiler to produce predictable parallelised code.

In structured grid computations, a finite many-dimensional data structure, known as the grid, is specified. The grid is parameterised by its dimensions and the type of its elements. Following Ypnos conventions, the identifiers X, Y and Z denote particular dimensions, \( \text{dim} \) represents a dimension identifier and \( D \) represents all dimension terms, or more formally:

\[
D := \text{dim} \mid D \times D
\]

In a function called \( \text{run} \), a stencil function is applied to each element of a grid in turn, whereby the element being operated on is referred to as the focal point. Typically a stencil function will operate on the focal point and some of the neighbours of the focal point.

\[
\text{stencil} : \text{Grid } D \alpha \to \beta \\
\text{run} : (\text{Grid } D \alpha \to \beta) \to \text{Grid } D \alpha \to \text{Grid } D \beta
\]
To specify a stencil function, Ypnos uses a system of *grid patterns*, whereby a programmer diagrammatically declares the cells a function operates on. Consider the following example of a grid pattern on a structured grid of dimensions \( X \times Y \). The @ symbol will bind \( c \) to the focal point of the grid. \( l, r, t \) and \( b \) will be bound to the elements to the left, right, top and bottom of the focal point.

\[
\text{ave2D} (X * Y): \quad \left| \begin{array}{c}
  \_ & t & \_ \\
  l & @ & r \\
  \_ & b & \_ \\
\end{array} \right| = (t + l + c + r + b) / 5.0
\]

As Ypnos enforces the single, independent writes property, there are no conflicts between writes therefore the run function can be parallelised.

This project aims to embed these features of Ypnos within another functional language.

### C.2 Resources Required

The project will extend an open-source functional interpreter, probably HaMLet [3]. The project will use this interpreter and its transitive resources, for example a parser generator. In order to add parallelisation to this interpreter PolyML [4] can be used, since it has concurrency support. Access is required to a multicore computer, ideally with eight processors.

### C.3 Starting Point

As well as the ability to program in ML, this project builds on the following courses:

**Concepts in Programming Languages** considers the module system of Standard ML, which is used heavily by HaMLet.

**Compiler Construction** will prove invaluable in understanding the workings of the interpreter.

**Concurrent and Distributed Systems** covered threads and the theory of producing code that is safe when run concurrently.

**Optimising Compilers** should enable the implementation to produce efficient code—a necessity for an efficient parallelised language.
C.4 Substance and Structure of the Project

The aim of the project is to pick a subset of Ypnos to implement as an embedded domain-specific language within a functional programming language. The precise syntax used when embedding Ypnos must be decided upon—this is likely to be very similar to the original syntax with language-based modifications. This project proposes to support grids of dimension \((X \times Y)\) to avoid problems in visually drawing grids of higher dimensions, and potentially extend if time allows.

Having clarified the language details, the implementation of Ypnos into an existing interpreter can begin. As a Ypnos compiler already exists, for Haskell, the structure of this can guide the development. The implementation consists of two parts; representing a grid and representing a Ypnos program. The proposed interpreter represents every program as an abstract syntax tree, therefore the AST must be modified so that it can also store Ypnos nodes.

The lexer may need to be modified. The parser will need to be extended to produce an Ypnos abstract syntax tree. The type-checker must be extended in order to ensure that all Ypnos programs are type correct and grids don’t contain any structures consisting of \texttt{ref} cells, as Ypnos’s parallelisation relies on grids being pure.

A backend will need to be created which will parallelise and execute the AST. Initially the backend will be written without any parallelisation, so that the parallelisation can be evaluated with a non-parallelised version. The backend will then be refactored to make use of the concurrent libraries in PolyML.

An evaluation can be performed by writing a program which will produce grids of a substantial size and then applying operations on these grids, measuring the run time costs of the operations. For example, Conway’s Game of Life could be used as a test program. Comparing the results from the final version with the results from the non-parallelised version will indicate how successful the parallelisation was. An interesting point of evaluation will be to compare the effect of parallelisation as grid sizes increase, especially when complex grid patterns are used. This can be extended to consider how
the number of cores used and number of threads per processor affect the performance of a Ypnos program.

The results can also be compared to the time the Haskell Ypnos compiler takes to perform the same computation and the time it would take to perform a computation in a functional language without Ypnos.

C.5 Variants

There is an element of choice as to the language in which Ypnos is both embedded and interpreted. As Ypnos is designed to be a pure functional EDSL, embedding Ypnos within a language in the ML family would be sensible, this could be Standard ML, Objective Caml or F#. Embedding Ypnos in other functional programming languages could also be successful.

It will be necessary to choose an interpreter to extend in order to add the functionality provided by Ypnos. The interpreter extended will depend on the availability of open-source interpreters for the language into which Ypnos is to be embedded. If SML is used then there exist a number of well-designed open-source interpreters, most notably HaMLet [3].

Adding concurrency to HaMLet can be done in a number of ways: if compilation is restricted to either PolyML [7] or SML/NJ [25] then the language’s built in concurrency packages can be used. The project could compile ML into C so that concurrency could be gained from pthreads or Cilk [26] whilst using the Ocaml compiler for the majority of the implementation. Also features could be removed from ML and then a backend could be written which compiles this subset into C. This would allow use of C’s Message Passing Interface [2], however this approach would make Ypnos programs which run simultaneously on multiple computers rather than on just one. CUDA could also be used in a compiled backend, to run the Ypnos program on a GPU however this is a NVIDIA proprietary language extension—something which is against the design principle of Ypnos.

Whilst it is vital that the type-checker is extended, extending the type-inference engine is not vital, instead grids can be annotated with their types.
C.6 Success Criteria

For the project to be deemed a success it is imperative that the objectives specified below are implemented:

1. An implementation of a \((X \times Y)\) grid data structure—a representation of a finite 2-dimensional discrete data structure.

2. An internal representation of a valid Ypnos stencil function must be designed and implemented.

3. A process of lexing and parsing a static Ypnos grid pattern to convert it into the format required for the internal representation of a Ypnos function must be implemented.

4. The interpreter must apply a stencil function to a grid and produce another grid containing the correct output.

5. The solution must produce parallelised code which runs on variable number of processors.

6. A detailed evaluation of the efficiency in parallelisation of the interpreter must be performed.

7. The dissertation must be planned and written.

C.7 Extensions

- Extend ML’s type inference so that grids do not have to be annotated with their types.

- Implement the reduction feature of Ypnos, whereby the application of a stencil function reduces the grid to a single value such as an average or deviation.

- Modify the implementation in order to remove the restriction of only allowing 2-dimensional grids to be entered, as is allowed in the Ypnos paper.
C.8  Timetable and Milestones

C.8.1  Preliminary Work

Decide on an interpreter to extend and finalise the way that concurrency will be implemented. Learn the structure of the interpreter which is being extended and the way in which PolyML supports concurrency.

Milestones: Be ready to begin coding.

C.8.2  Weeks 1 and 2

Decide on the syntax to be used for Ypnos by producing Backus Normal Form. Modify the lexer and begin work on extending the parser.

Milestones: All preparatory work is complete. A formal definition of the syntax is decided upon. The parser is partially complete.

C.8.3  Weeks 3 and 4

Finish work on the parser. Begin updating the type-checker.

Milestones: The interpreter can input ML with embedded Ypnos and produce an abstract syntax tree. The type-checker is partially complete.

C.8.4  Weeks 5 and 6

Complete the type-checker.

Milestones: The interpreter should build a correct abstract syntax tree for any correct program with embedded Ypnos and give an error if any incorrect program is entered.

C.8.5  Weeks 7 and 8

The programming of a non-parallelised version of the backend will begin.
C.8.6  Weeks 9 and 10

The non-parallelised backend of the interpreter will be finalised and tested.

Milestones: A fully functional Ypnos interpreter should now be written which will correctly run Ypnos programs.

C.8.7  Weeks 11 and 12

Write a harness of programs which can be used to evaluate the interpreter. Begin work on parallelising the interpreter to fully utilise the benefits of Ypnos, using the concurrent support of ML.

C.8.8  Weeks 13 and 14

Finish the parallelisation of the backend of the interpreter. Start evaluating the implementation of Ypnos by building a program to generate large grids which stencil functions can be applied to.

Milestones: A final version of the Ypnos interpreter.

C.8.9  Week 15 to 20

Continue to evaluate the performance of the interpreter and complete the associated dissertation.

Bibliography


