CAST
A Parallel Programming Framework

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor - Master of Technology (Dual Degree)

by

Vikas Kushwaha

Supervised by

Prof. R. K. Ghosh & Prof. Harish Karnick

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2013

CERTIFICATE

It is certified that the work contained in the thesis titled **CAST : A Parallel Programming Framework**, by **Vikas Kushwaha** (**Y8127563**), has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. R. K. Ghosh
Department of Computer Science & Engineering
IIT Kanpur

Prof. Harish Karnick
Department of Computer Science & Engineering
IIT Kanpur

June, 2013

ABSTRACT

Name of student: **Vikas Kushwaha** Roll no: **Y8127563**

Degree for which submitted: **Bachelor - Master of Technology (Dual Degree)**

Department: **Computer Science and Engineering**

Thesis title: **CAST : A Parallel Programming Framework**

Thesis Supervisors: **Prof. R. K. Ghosh**, **Prof. Harish Karnick**

Month and year of thesis submission: **June, 2013**

Multi-core architectures and distributed systems have sparked a renewed interest in parallel programming. Thankfully, we have a range of programming abstractions such as message passing, software transactional memory, dataflow programming amongst others that allow us to write elegant solutions to parallel programming problems in their respective niche domains. Traditional languages based on the von-Neumann model of computation are inherently sequential and make heavy use of side-effects in their computations. It makes them unsuitable for accommodating such abstractions in a straight forward way.

We need programming languages that can provide a collection of abstractions such that we can choose the right abstraction depending on the problem. However, by merely adding the abstractions without any underlying framework, we run the risk of making the language complex and difficult to master.

In our thesis, we propose a parallel programming framework based on graphs that aims to accommodate various parallel programming abstractions while remaining simple and succinct. Then, we define a programming language based on the framework and develop an interpreter for it in Haskell. We demonstrate the usefulness of the framework, by providing programs to a range of problems involving parallelism, concurrency and distributed execution.

Acknowledgements

I would like to thank my thesis supervisors Prof. R. K. Ghosh and Prof. Harish Karnick for providing me with an opportunity to work on a problem in my area of interest. Right from the start, they allowed me the freedom to explore the problem as desired and flexibility to work at my own pace. This thesis would not have been possible, without their invaluable suggestions and constant encouragement.

I would also like to thank Prof. Sanjeev K. Aggarwal, for directing me to Prof. R. K. Ghosh, thereby helping me in finding a thesis topic in my area of interest.

Finally, I am grateful to the department of Computer Science & Engineering and IIT Kanpur, for providing me with an excellent environment for learning. These five years at the campus, have taught me a lot and provided me with experiences to cherish forever.

To my Parents and my Sister

Contents

List of Figures **xiii**

List of Programs **xv**

1 Introduction **1**

2 CAST Abstractions **3**

2.1 Operations 5

2.1.1 Cloning 6

2.1.2 Association 7

2.1.3 Specification 7

2.1.4 Translation 7

2.2 Features and Comparison with Existing Abstractions 9

2.2.1 Message Passing 9

2.2.2 Transactions 10

2.2.3 Object Oriented Programming 11

2.2.4 Functional Programming 11

2.2.5 Monads 12

2.2.6 Constraints and Properties 13

2.2.7 Dataflow and Reactive Computation 13

2.2.8 Homoiconicity 14

2.2.9 Type System 14

3 JackFruit : A Language Prototype **17**

3.1 Grammar 17

3.2 Description 19

3.2.1 Basic Building Blocks 19

3.2.2 Basic Associations 21

3.2.3 Message passing Associations 23

3.2.4 Specification and Cloning 23

3.2.5 Translation 24

3.2.6 Syntactic Extensions 25

3.3 Implementation 26

4 Sample Programs **29**

4.1 Pipelining: Sieve of Eratosthenes 29

4.2 Fork Join Parallelism : Fibonacci using Recursion 31

4.3 Dataflow Parallelism : Fibonacci using Dynamic Programming 31

4.4 Reactive Programming : Spreadsheets 31

4.5 Concurrent stateful modifications 32

4.6 Property Based Parallelism : Addition Reduction 33

4.7 Data Parallelism : Matrix Multiplication 33

4.8 Distributed Computation : MapReduce 33

5 Conclusions and Future Work **35**

5.1 Future Work 35

5.1.1 Complete Implementation of CAST Framework 35

5.1.2 Support for Pattern Matching 35

5.2 Conclusions 36

References **37**

List of Figures

2.1 An array of size 5 modelled as a graph 5

2.2 An example record modelled as a graph 6

2.3 A graph showing $d = a + b - c$ using data-flow and change-dependency edges 6

2.4 Entity for computing Fibonacci numbers 8

2.5 Computing 3rd Fibonacci number 9

2.6 Message passing in CAST 10

2.7 Implementing increment function in terms of add 12

List of Programs

4.1 Sieve of Eratosthenes . . . 30

4.2 Calculating Fibonacci numbers using recursion . . . 31

4.3 Calculating Fibonacci numbers using dynamic programming . . . 31

4.4 Spreadsheet programming . . . 32

4.5 Concurrent stateful modifications . . . 32

4.6 Addition Reduction . . . 33

4.7 Matrix Multiplication . . . 34

4.8 A MapReduce program with 3 compute nodes . . . 34

Chapter 1

Introduction

Most of the mainstream programming languages are based on the von Neumann model of computation, which presents an inherently sequential model of execution. In order to increase performance, we rely on increasing the processor clock speed. But, the above method is limited by, heat thresholds on semi-conductors. To overcome this problem the hardware community has come up with multiple core processors. In parallel on the software front, the trend is to move away from a single powerful server model in favour of distributed systems that provide better performance and reliability.

There is, thus, a need to write applications aimed at capturing parallelism offered at the system level. Traditional languages such as C, C++, Java amongst others based on the von Neumann Model are sequential in nature, with almost no support for parallelism at the abstraction level. Parallel programming libraries and extensions based on such languages are cumbersome to use and require the programmer to focus on low level details of data-races and scheduling. Furthermore, the code produced becomes hard to debug and understand. Consequently, it poses non-trivial challenges for writing large software using these tools.

The need of the hour, is thus a programming model, which allows us to express parallelism in an intuitive and declarative fashion, while incorporating other notable features such as modularity, simplicity in organization and retrieval of data, incremental design and homoiconicity.

Above mentioned features have advantages of their own. Modularization provides us the ability to express our solutions in terms of small logically separated modules, while presenting relationship between them explicitly. In fact, the more we are explicit about the relationships between modules the better are our chances to use that information to parallelize our code. For example, use of pure functions as building blocks in functional languages opens possibilities for parallelization based on explicitly available information on data dependency. Another aim of modularization is to obtain a separation of concerns between design and implementation details. Incremental design enables programmers to write the first version of the software quickly, which can then be improved further into a full fledged system by adding features gradually. Homoiconicity allows us to extend our language with new concepts very easily. Also, it results in an elegant, succinct and simple language as the same operations can be used to define and modify both code and data. A tasteful combination of the above features and parallelism is required to create a good programming language.

As a part of this thesis, we have developed a parallel programming framework (called CAST). Then, we designed a language prototype (named JackFruit), which provides an implementation of the framework. Finally, we developed an interpreter for the language prototype using Haskell.

The rest of the thesis is organized as follows.
Chapter 2 presents the CAST framework, discusses its features and compares them with existing abstractions.
Chapter 3 describes the syntax and semantics of the JackFruit language, with an overview of its implementation.
In **chapter 4**, we present a collection of problems from the combined domain of parallel, concurrent and distributed computations, and provide solutions to them using JackFruit.
Finally in **chapter 5**, we conclude by describing possible future work.

Chapter 2

CAST Abstractions

As the current state of work in programming languages stands, we have a collection of abstractions and features, each providing elegant solutions to problems in their niche domains. Message passing, software transactional memory, object oriented programming, functional programming, imperative programming and reactive programming are some of the notable examples. Every practically useful general purpose programming language is expected to provide support for these well known features in their syntax or emulate them through external libraries. With the first alternative, we fear increasing the complexity of the language. Choice of second alternative, depends on the quality of emulation. An implementation, difficult to use or one which defeats concise representation, proves as a bottleneck. The solution for the first alternative is to use a clever syntax which accommodates most of the abstractions easily. This approach has been taken by languages such as Scala, Python and Ruby. Another solution is to develop a generic framework, such that these abstractions can be defined as special cases of it. For example, traditional imperative programming can be considered a special case of monadic framework. Lisp is another example which with its list based homoiconic representation and macros, provides a framework, to define new abstractions easily.

We believe that language syntax is easier to remember and understand, when it is based on some underlying framework. Hence, we take the generic framework based approach for our programming language. The framework developed by us, has graph at its core. Motivation for this decision, is based on the following ideas.

- Graph is very well suited to represent dependencies and relationships between objects.

Dependencies, play a very important role in parallelization.

- A wide range of data structures, including array, linked list, record, heap and binary search tree can be modelled as graphs.

The CAST framework consists of a set of four operations on a basic design unit called *entity*. An entity is defined as an active heterogeneous graph, which is used to represent both computation and data. It, like any other graph, consists of two types of components: nodes and edges.

Node: A node represents the basic entities, which act as building blocks for other entities. An implementation is free to choose its own set of basic entities (examples - character, integer and string literals, identifiers etc).

Edge: An edge represents an active or waiting computation involving the two entities that it connects. A waiting computation is one that is waiting on some condition involving the concerned entities becoming true, before it becomes active. Computations also have a lifetime associated with them. Some computations such as the ones representing a change-dependency remain alive forever, switching alternately between waiting and active states, in response to changes. There are others such as message passing which finishes once the message has been sent. Assignment is another such computation, which ends once the assignment is done. Because an entity can have many edges incident on it, the active phase of a computation defined by an edge, must complete atomically, to maintain consistency. Starting with known entities, edges can also be used to locate other entities in the graph. Note that an edge is also an entity, and hence, can be defined in terms of already existing entities and edges. (See fig. 2.1, 2.2, 2.3 and 2.6 for examples of edges)

Use of graph edges to represent computations has been motivated by following examples.

- Assignment operation in many of the imperative languages takes the form `A := B`.
- On a similar note, equality assignment operation takes a form like `A = B` (A is always equal to B and vice-versa).
- Change dependency is an operation involving two entities, such that change event for one entity, generates a change event for the other.

- Message send can be represented similar to `A => B` (message A sent to B).
- Message receive can take the following form : `A <= B` (message received from B, is referenced as A).

Note that, the form taken by above operations is similar to two entities, connected by an edge. Further, these operations when used in combination with other operations provided by the hardware (e.g. addition, multiplication etc) yield different computation models. For example, assignment together with basic arithmetic forms the core of imperative programming. Functional programming relies on equality assignment operation to define functions. Change dependency allows us to write reactive computations. Send and receive operations form the core of message passing model.

Since, a sub-graph is also a graph, any part of the entity graph is also an entity in itself. We mentioned that an entity graph is heterogeneous because, different parts of a graph can have different implementations (see translation operation in section 2.1.4). Starting with basic entities, we can define and evolve new entities using four operations, namely cloning, association, specification and translation.

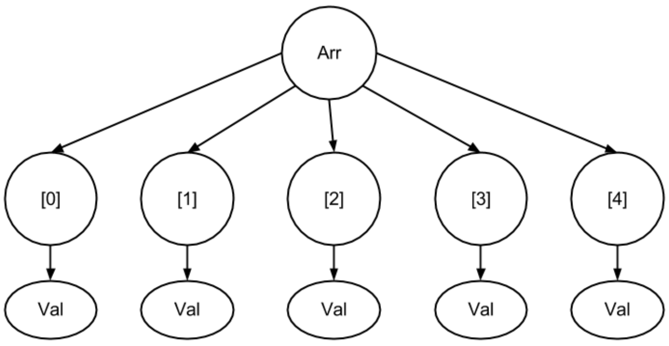

Figure 2.1: An array of size 5 modelled as a graph

2.1 Operations

CAST framework consists of four operations. Cloning and association form the basic operations required to create a graph. Specification provides for graph modification. One of our initial aim was to provide a separation of concerns between program logic and implementation details. Translation provides exactly that. It frees us from restricting ourselves to a particular graph implementation. Instead, we can switch implementation

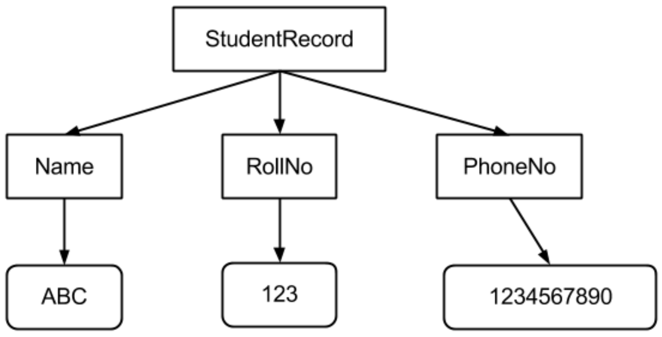

Figure 2.2: An example record modelled as a graph

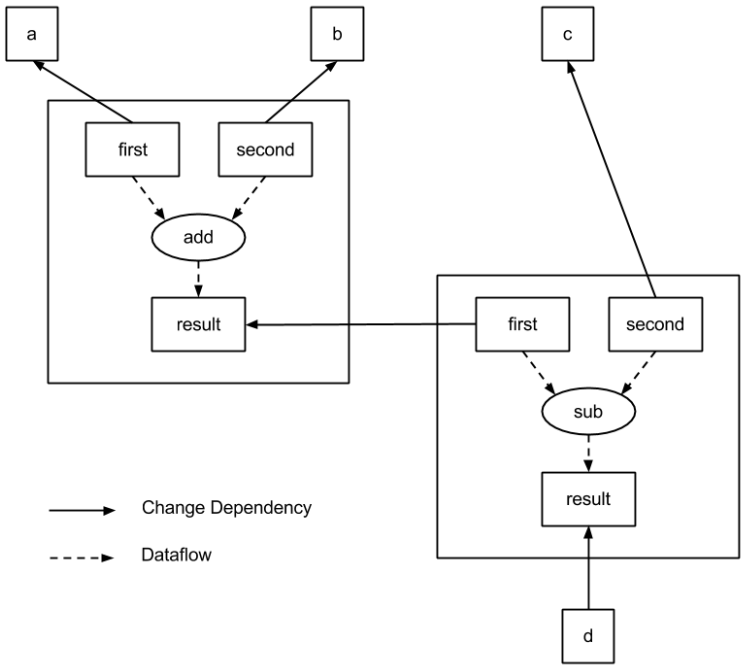

Figure 2.3: A graph showing $d = a + b - c$ using data-flow and change-dependency edges

as desired, without affecting the program logic. Closer analogues of translation in other languages are : interfaces in Java and function pointers in C. We will now describe these operations in detail.

2.1.1 Cloning

Cloning creates new copies of existing entities. If we have to clone an entity, which is not in a quiescent state, we wait for it to become quiescent, before proceeding. (See fig. 2.4 and 2.5 for example).

2.1.2 Association

Association is the edge creating operation. It takes three entities, one of which is the edge entity and connects the other two. Association is not allowed as a stand-alone operation and must always be a part of some specification. It might be possible that the two entities connected by the edge are not in the desired state as required by the computation represented by the edge. In that case, it is the duty of the edge to wait as desired, before achieving active phase. The task of association is only to create the edge, which then executes on its own. Together, with cloning and association, we can compose new entities out of existing ones. (See fig. 2.4 and 2.5 for example).

2.1.3 Specification

Specification is an operation composed of a collection of cloning and association operations, which modifies an entity asynchronously and atomically. The model does not enforce any ordering on the operations contained in the collection, except ones due to dependencies. As a result, it is possible that different orderings may lead to different results. It is upto the implementation to enforce any constraints on the ordering or contents of the collection. Since, multiple specifications can act concurrently on an entity, atomicity condition allows us to ensure consistency. Specification allows code re-use, as we can create a new entity by cloning an existing one and then modifying it. (See fig. 2.4 and 2.5 for example).

2.1.4 Translation

While writing programs for solving graph theoretic problems, we choose a data structure to represent the graph : most common options being adjacency matrix and adjacency lists. Here note that the program logic is independent of the representation we use, since graph is defined abstractly and its definition is independent of the representation. Translation achieves something similar.

Translation takes two entities, one providing the description and other providing the implementation and then lays out (executes/synthesizes) the description based on implementation. Description entity describes how the resulting graph entity must look like while the implementation entity gives us one of the possible ways to build such a graph entity. For example, entity representing a collection of numbers can either be implemented using an array or a linked-list. A function written in C language can be viewed as a description entity, which is laid out on the stack, when executed. A file is an entity that can be synthesized on a file-system. Any specification request to description entity is transformed into operations determined by the implementation entity. For example, read/write operations on file entity are translated into instructions for hard-disk driver. An implementation entity is free to provide definitions for only a subset of entity operations and built-in entities. In that case, specifications resulting in non-defined operations will fail. For example, a linked-list entity can provide implementation for a linear graph. Now, if we try to do a specification on this linear graph, which makes it non-linear, this will result in an exception as our linked-list implementation cannot support non-linear graph. Similarly, a struct in C language can be used to implement a restricted graph. (See fig. 2.4 and 2.5 for an example).

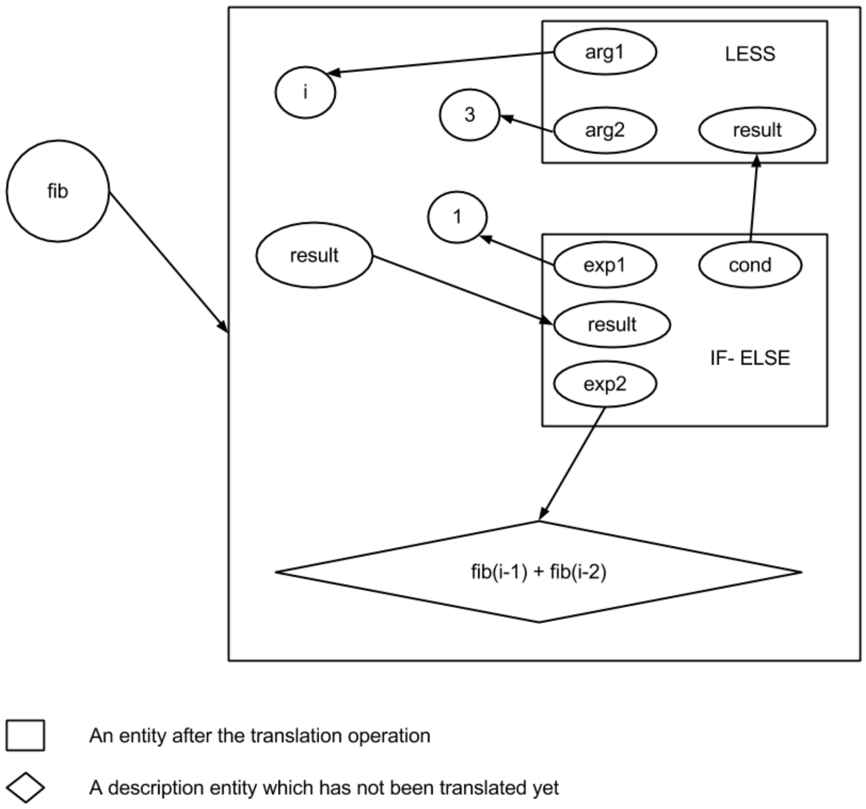
arg1
LESS
i
3
arg2
result
fib
1
result
exp1
cond
result
IF- ELSE
exp2
fib(i-1) + fib(i-2)
An entity after the translation operation
A description entity which has not been translated yet

Figure 2.4: Graph shown in this figure represents an entity for calculating Fibonacci numbers. Mathematically, `fib(i) = if i<3 then 1 else fib(i-1) + fib(i-2)`

Next, we will examine features of CAST framework and compare them with existing abstractions as we go along.

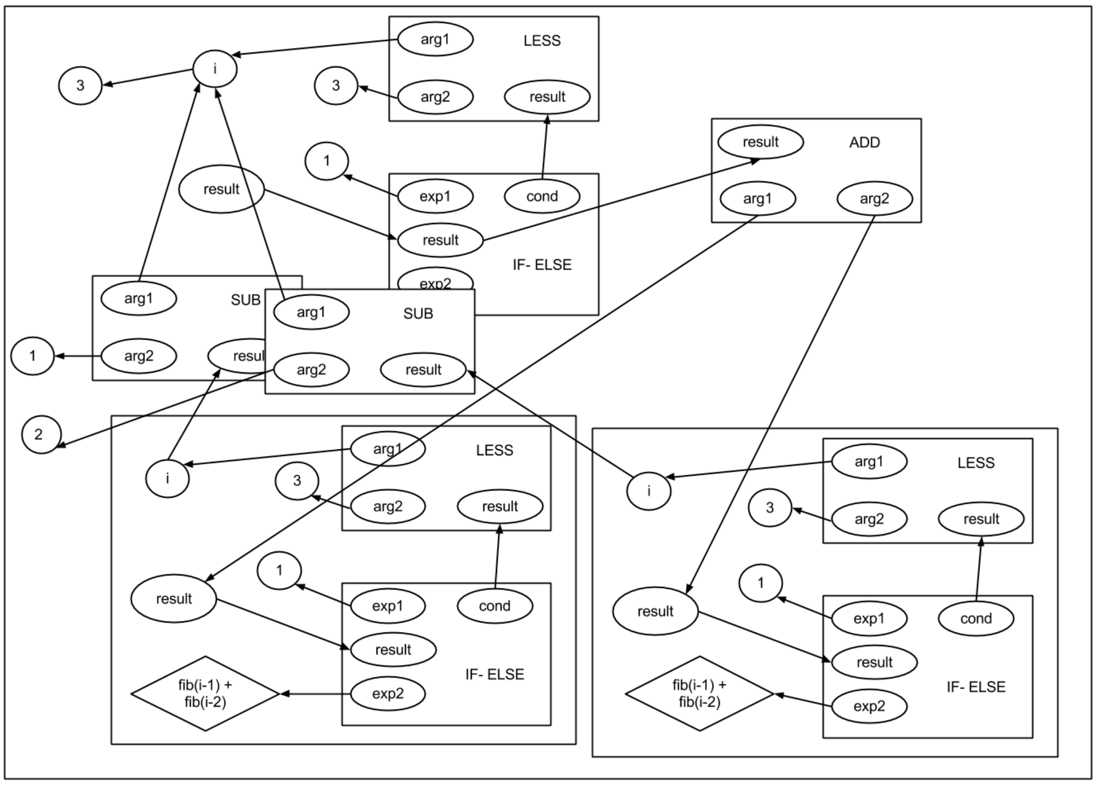

Figure 2.5: In the this graph, we first create a clone of fib entity defined in fig. 2.4. Then, we do a specification on the cloned entity by associating i with 3. Bounding i results in the activation of if-else, which selects the description entity linked with exp2 and translates it into the corresponding graph using the default implementation.

2.2 Features and Comparison with Existing Abstractions

2.2.1 Message Passing

Message passing is a form of communication where processes or objects can send/receive messages (comprising of data or maybe even segments of code) to other processes or objects. It can also be used to control access to resources in a concurrent system, the other alternative being locking.

Actor Model [1] is a one such message passing system. It provides us a way of solving problems which can be modelled as a number of independent processes which interact with each other only through message passing. It can be used to model a wide range of concurrent and distributed systems. However, it is terrible at implementing a truly shared state [2]. For example, consider the case of a banking problem of money transfer from one account to another. We require that this transfer should be done atomically, which is difficult to realize if these accounts are represented by actors.

In CAST, message passing can be envisioned as a process involving association between the message and the buffer entity on receiver side. The message entity dissociates itself with its parent entity during the send process (refer to fig. 2.6). Further, a change dependency on message buffer, can be used to write code that reacts to new message received.

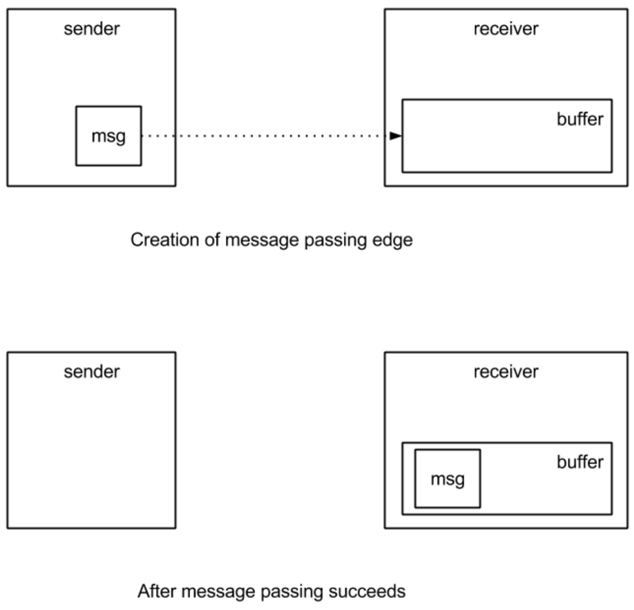
sender
receiver
msg
buffer
Creation of message passing edge
sender
receiver
msg
buffer
After message passing succeeds

Figure 2.6: Message passing in CAST

2.2.2 Transactions

Composable transactions provide a good solution to the banking problem discussed in previous section. Lock based solutions require us to think about overlapping operations between seemingly unrelated areas of code, which is a difficult and error prone task. While using transactions, this task is done automatically for us. If two transactions executing concurrently try to update a common state, only one of them completes, while the other either waits or is restarted. Software transactional memory allows us to implement transactions without losing on parallelism. STM itself can be implemented either through lock based memory or versioning based memory.

To get an equivalent of transactions in CAST, we can just use a specification, which guarantees atomicity.

2.2.3 Object Oriented Programming

In OOP, program can be viewed as a collection of stateful interacting objects, which can send/receive messages and perform operations which are closely associated with the object itself. Objects combined with inheritance provide us a good way to model certain class of problems where object has preference over operations and operations can be clearly articulated. Examples are GUI systems, databases etc. The model does not fits well in cases where no compelling reason exists to prefer the object over the operation involved. For example, consider this computation : calculate `a + b`, where a, b are integers. When we try to fit this problem to the OO world, we need to choose one of a and b as the main object, so that '+' can be implemented as a method. It looks something like this : `a.add(b)` or `b.add(a)`. Now, we face a dilemma of choosing one form over the other, with no clear winner. Object, such as databases generally have a established set of operations that can be performed on them. However, it would be nearly impossible to articulate the set of all operations that we might perform on integers. Alternatively, by giving the '+' operation importance over data involved, we could write the computation simply as : `c = add(a,b)`.

Since, entities can be a combination of both code and data, they can model objects very easily. On the top of that, using a combination of cloning and specification, we can define new entities out of existing ones in a dynamic and incremental fashion. We can both add and delete features. Class based inheritance, on the other hand, is restrictive and problems such as ellipse-circle relationship are difficult to model with it [3]. In CAST, we can define circle as a specified clone of ellipse, with `a = b = r` and `a`, `b` made non-editable. Prototype based object oriented languages such as Self, however omit classes, instead they use cloning as a alternative to inheritance. As a result, they don't suffer from restrictions present with class based inheritance.

2.2.4 Functional Programming

Functional Programming Languages are generally considered as extensions of lambda calculus. An equivalent of a function can be implemented in CAST, by allowing for place-holder entities, which can be replaced by entities representing function parameters, when a function entity is invoked (or in terms of CAST: specified). Since, entities can be created dynamically, we could very well define lambda functions. Partial functions can be defined using a combination of cloning and a specification which does not provide all the function parameters (see fig. 2.7).

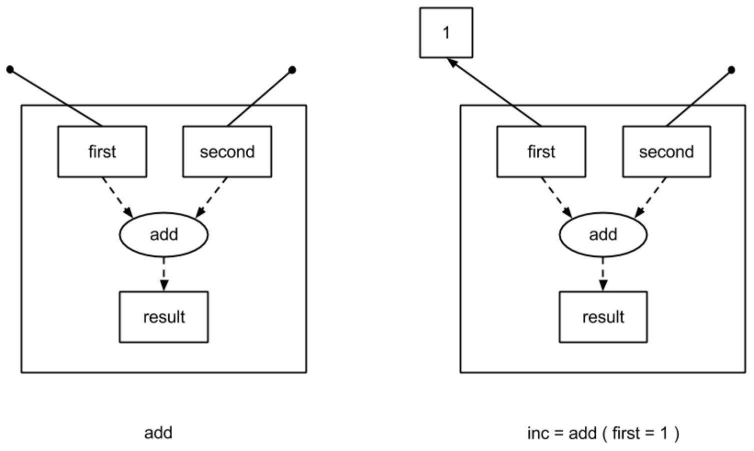

Figure 2.7: Implementing increment function in terms of add

2.2.5 Monads

In imperative programming languages, we transfer the full state of the program and associated environment between consecutive statements. This means that every statement is made aware of the changes made by the previous statements. In pure functional languages, no contextual information is transferred between two function invocations. Hence, various function invocations are only linked through data dependencies between them, providing various opportunities for parallelization. However, a large collection of programming concepts are imperative in nature such as state manipulations, concurrency, exception handling , input/output etc. Monads offer us a solution such that both ways of programming can co-exist. It does so by allowing us to be explicit about the context information passed between two statements [4]. In terms of imperative languages, we can view it as a ”programmable semi-colon”. In terms of functional programming, we view it as extra parameter passed between functions. Based on the type of context information being transferred, compiler can take better decisions regarding parallelization and other optimizations.

For passing context information around entities in CAST, we can use edges programmed to transfer different types of contexts. For example, we could define an edge, which presents the equivalent of a semi-colon in imperative languages, resulting in sequential execution of the entities involved.

2.2.6 Constraints and Properties

Constraints are edges which monitor the value of concerned entity, such that it passes the given condition. In case it fails, an exception is raised.

Some characteristics of an entity cannot be deduced from its definition. For example, we cannot easily determine associativity and commutativity of addition through its definition. But, if known, one can utilize this information to parallelize addition of collection of numbers. To address this, we can associate addition entity with property entities representing associativity and commutativity. A property entity can represent a concept as a string identifier, which is not obvious to infer otherwise.

2.2.7 Dataflow and Reactive Computation

Dataflow systems model program as a directed graph specifying flow of data between operations. Operations act as black boxes which automatically run as soon as input data is available. This model is well suited to express computational dependencies between data. Since, the dependencies are known, the non-dependent portions of the graph can be executed in parallel. By making the graph dynamic, we can add more computations by introducing new nodes and edges.

Reactive programming is an event based model which finds interesting applications in animations, GUIs, spreadsheets, hardware description languages etc. It involves defining a data dependency, such that any changes are automatically propagated through data dependency flow.

Its very natural to express dataflow and reactivity in CAST, as these are simply edges in entity graph. For an example refer to figure 2.3.

2.2.8 Homoiconicity

Meta-programming is the ability to write programs which take other programs as their data and manipulate them. The language of the meta-program is called meta-language. The ability of a programming language to be its own meta-language is called reflection. Having the programming language itself as a first class data structure is known as homoiconicity. Lisp [5] is an example of homoiconic language. The main advantage of such a language is its succinctness and simplicity. Adding new concepts becomes simpler, as code can be manipulated in ways, a normal data structure would be manipulated. Most of the existing homoiconic languages are based on the list data structure. This presents a disadvantage, as by doing so, it does away with many of the visual cues that help humans visually parse the language. It results in steep learning curve for the language [6].

CAST framework is homoiconic with graph acting as the base data structure. It overcomes the lack of representational ability of list, thereby removing the disadvantage discussed in previous paragraph. In fact, graph is very closely associated with how human think in terms of concepts and relationships between them. A list can be viewed as a linear graph, while it is difficult to represent a graph in terms of a list. Graph is a mathematical structure with many possible implementations. Further, the choice of implementation depends upon the set of graph operations required for the particular use case under consideration. Using the translation abstraction, CAST allows us to switch the implementation, while retaining the same syntax for graph operations. For example, suppose we need to create a collection of integers. In CAST, we can represent the collection as a linear graph. This graph can be implemented using a linked-list or an array. CAST allows us to switch implementation from linked-list to array and vice-versa, while we can operate on the collection using the same graph operations. This is a contrasting feature as compared to Lisp like languages, where the implementation of list is fixed by the interpreter/compiler.

2.2.9 Type System

A type system has two purposes. First is to associate constraints and properties with a given object. This is to convey important information which determines program logic and in other cases, detects error cases in timely manner. Second purpose is to convey implementation information, which though does not affects program logic, affects time and memory complexity of the program. For example, `long` and `int` types in C language denotes that the object under consideration is integral and declares the number of bytes to use for implementation. CAST serves the first purpose by the use of constraints and the second through translation. Both of them have been discussed in previous sections. As an example, consider an integral entity that fits within 2 bytes. In CAST, we can associate the entity with constraint that it should be integral, and using translation, specify that it should be implemented using a two byte memory chunk.

While static type system is necessary to enforce program correctness and reduce run-time errors, it makes incremental design difficult by rejecting partially correct programs. Dynamically typed languages (also called Uni-typed languages, as a single type is used for everything), on the other hand, allow us to write first version of a program fairly quickly. There is a trade-off between the development time versus program efficiency and correctness, when choosing one over the other. The solution here is optional typing, which allows programmer to add type information incrementally. The runtime uses the boxed type representing all types, when type information is not provided. Several languages already have optional type annotations. Examples include Common Lisp, Dylan, Visual Basic.NET etc [7]. In CAST, we can have optional typing by providing a default graph implementation which provides for all graph operations.

Chapter 3

JackFruit : A Language Prototype

In this chapter, we describe the *JackFruit* programming language, which serves as a prototype for a concrete programming language based representation of the CAST framework.

3.1 Grammar

Notation The following grammar uses BNF representation. $*$ denotes zero or more occurrences. + denotes one or more occurrences. [] are used to denote an optional element. <> represent non-terminals and boldly printed words denote terminals.

$\langle program \rangle$ ::= $\langle stmt \rangle$*

$\langle stmt \rangle$ ::= $\langle association \rangle$

$\langle association \rangle$ ::= $\langle transmit \rangle$
| $\langle expr \rangle$
| `remove` $\langle name \rangle$
| $\langle dependency \rangle$
| $\langle assignment \rangle$
| $\langle assignment\text{-}sync \rangle$
| $\langle receive \rangle$
| $\langle property \rangle$
| `forEach` $\langle name \rangle$ `in` $\langle expr \rangle$ `weHave` $\langle stmt \rangle$
| `when` $\langle expr \rangle$ `weHave {` $\langle stmt \rangle$* `}`

$\langle transmit \rangle$ ::= $\langle expr \rangle$ `=>` $\langle expr \rangle$

```
⟨dependency⟩        ::= ⟨name⟩ = ⟨expr⟩

⟨assignment⟩        ::= ⟨name⟩ |= ⟨expr⟩

⟨assignment-sync⟩   ::= ⟨name⟩ := ⟨expr⟩

⟨receive⟩           ::= ⟨name⟩ <= ⟨expr⟩

⟨property⟩          ::= : ⟨name⟩

⟨expr⟩              ::= ⟨base⟩ ⟨specification⟩*
                      | ⟨undefined⟩
                      | ⟨if-else⟩
                      | ⟨translation⟩

⟨translation⟩       ::= ⟨expr⟩ @ ⟨expr⟩

⟨base⟩              ::= ⟨integer⟩
                      | ⟨character⟩
                      | ⟨boolean⟩
                      | ⟨string⟩
                      | ⟨qualified-name⟩

⟨qualified-name⟩    ::= ⟨name⟩ [.⟨name⟩]*

⟨specification⟩     ::= ⟨result-spec⟩
                      | ⟨entity-spec⟩
                      | ⟨change-spec⟩

⟨result-spec⟩       ::= ( ⟨stmt⟩* )

⟨entity-spec⟩       ::= [ ⟨stmt⟩* ]

⟨change-spec⟩       ::= { ⟨stmt⟩* }

⟨undefined⟩         ::= ?

⟨if-else⟩           ::= if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩

⟨name⟩              ::= ⟨identifier⟩
                      | ⟨int-list⟩

⟨int-list⟩          ::= < ⟨expr⟩+ >
```

3.2 Description

Note : A program in JackFruit is treated as a specification on a clone of empty entity. Also, collection of operations making the specification are executed in order in which they are stated.

3.2.1 Basic Building Blocks

Comments

The language currently supports only one line comments starting with a `#` symbol.

Delimiters

Except for the use of *space* for separating tokens the language has no delimiters. However, a programmer may use comma, semi-colon or newlines, anywhere they wish to improve readability. They are ignored by the parser.

Literals

Integer, Character, Boolean, String literals represent entities that cannot be specified (changed) later. Syntactically, they are represented in a manner similar to C.
Examples : `123, -78, 'a', true, false, "string"`

Builtins

Builtins are entities which are provided by the runtime. System calls provided by the operating system and facilities provided directly by the hardware, are generally provided as builtins. Only limited functionality is exposed to the programmers. The actual implementation is hidden. They are used as black boxes in programs. Examples are entities for addition and subtraction operations.

```
# builtin entity sub appears like this to the programmer
sub = $[ x = ?
         y = ?
         $result = ...
       ]
# the implementation defining the relationship
# between x, y and $result is not visible.
```

Undefined Symbol

An undefined symbol is denoted by `?`. It is used as a place-holder entity. A non-associated entity passed during specification, replaces the first place-holder it finds. This allows us to write a specification with a syntax similar to function invocation. The example below makes the idea clearer.

```
add = $[ x = ?
         y = ?
         $result = $plus(x, y)
       ]

inc = add[1]  # is equivalent to add[x=1]
inc2 = add[y=1]
r1 = inc(5)   # 5 will bind to y
r2 = inc2(8)  # 8 will bind to x
print r1  # will print 6
print r2  # will print 9
```

Empty Entity

It is the starting point for creating a specifiable (modifiable) entity. New specifiable entities are created by either specifying an empty entity, or by specifying an existing specifiable entity.

Name Entity

A name entity holds a reference to another entity. A name can either be an identifier or a list of expressions which evaluate to integers. Identifiers can start with either an alphabet or $ or %, and continue with alphanumeric characters. Name $ represents the empty entity. Integer list based names are used as an analogue of array indexes in other languages. Array indexes are also treated as names in JackFruit.

Examples: `var`, `<1,2>`, `<add(i,1), sub(i,1)>`

Qualified Names

A list of names separated by dots. A qualified name `A.B` is decoded as follows: Suppose `A` is a name entity and refers to some non-name entity (say `C`). Now, we search for a sub-entity inside `C`, which represents name entity `B`. This sub-entity is what we are looking for. If no such sub-entity exists, then we explore the other end of all the edges emerging from `C` for a name entity denoting `B`.

Examples `student.name.surname`, `arr.<1,2>` is an analogue to `arr[1][2]` in C.

Properties

Properties are identifiers starting with a colon. They are used to annotate extra information associated with an entity. The following example explains it further.

```
add = $ [ x = ?
          y = ?
          $result = $plus(x,y)
          :associative
          :commutative
        ]
```

3.2.2 Basic Associations

To describe the semantics of association operations, we need to define the function `valueOf`

valueOf : `valueOf(E)`, where E is an entity, is defined as equal to `valueOf(E')`, when `E` is a name entity which refers to `E'`. In other cases, `valueOf(E)` is equal to `E` itself.

Change Dependency

`A = B` creates an edge between `A` and `B`, such that `valueOf(A)` is defined to be `valueOf(B)`. The resulting edge has an infinite lifetime, and can only be explicitly removed. Further, any change in `valueOf(B)` results in an event signaling change in `valueOf(A)`. A change dependency edge can lead to circular dependencies. For example, `A = add(B,A)` results in a circular change dependency and `valueOf(A)` cannot be determined.

```
a = 8
b = a
c = b
print c  # 8
a = 9
print c  # 9
b = 10
print a  # 9
print c  # 10
```

```
x = 3
y = x
z = add(y,1)  # z has value 4
x = 5   # z has value 6
```

Assignment

`A |= B` results in a temporary edge between `A` and `B`, which waits until `valueOf(B)` becomes determinate (not unbound), after which, it binds `A` with `valueOf(B)` and removes itself. It is the equivalent of an asynchronous assignment operation in imperative languages.

```
a = ?
b = a
c |= b
a = 8
print c  # 8
a = 9
print c  # 8
```

Synchronized Assignment

Synchronous version of Assignment. Instead of waiting for `valueOf(B)` to become determinate, it successfully executes if `valueOf(B)` is determinate at the edge creation time and fails otherwise. It is denoted by `:=`

```
b = ?
c := b  # cannot fructify as b is non-determinate at edge creation time
b = 8
print c  # <unbound>
```

Remove

`remove name` dis-associates the entity represented by `valueOf(name)` of all its existing associations with any sub-entity of the entity being specified.

```
s = $[ x = 5
       y = 10
     ]
s { remove x }
# after above operation, s becomes
# $[ x = ?  y = 10 ]
```

3.2.3 Message passing Associations

Transmit

`A => B` attaches entity represented by `valueOf(A)` to the end of entity represented by `valueOf(B)`, while dis-associating `valueOf(A)` from its existing associations. End here means, that the added entity should come last, if one iterates over the entities of `valueOf(B)`.

```
data = $[1,2,3,4,5]
add(5,1) => data
# data contains [1,2,3,4,5,6] at the end of operation
```

Receive

`A <= B` will bind `A` with the first entity of `valueOf(B)` and dis-associate the first entity from `valueOfB`.

```
data = $[1,2,3]
x <= data
print x  # prints 1
```

3.2.4 Specification and Cloning

Specification is a collection of statements, comprising of cloning, association and other specification operations, that executes atomically and asynchronously. Statements inside the specification operation, share the namespace of both the entity to be specified and also the entity from which the specification has been made. This could result in a clash between similarly named entities from the two namespaces. We resolve this clash, in favour of the entity to be specified. To denote a clashing name belonging to an entity from which the specification has been made, we append a back-quote before the name. The following example clarifies this further.

```
x = 5
f = $[ x = 6 ]
f { y = x }  # x here refers to f.x
f { y = ‘x }  # x here refers to the one in first line
```

JackFruit has three syntax variants to express specifications.

Result Specification

It is represented by a sequence of statements between enclosing `()`. Cloning is also implicitly performed with this operation. Output of this specification is the entity represented by `$result`, present inside the new clone entity formed.

```
z = add(x, y)
# create a clone for add (say add'), bind x and y as parameters for add',
# then finally bind z to add'.$result

s = $[ x = ?
       y = ?
       $result = mul(x, add(y,1))
     ]
t = s( x = 5   y = 1)       #  t has the value 10
```

Entity Specification

It similar to the result specification except that the output is the cloned entity itself. `[]` acts as the en-closer.

```
x = fibonacci[n = 10]
# x does not represents the 10th fibonacci number, instead
# it represents the fibonacci entity created to compute the 10th fibonacci number
print x.<10>  # prints 10th fibonacci number
```

Change Specification

It is similar to the entity specification except that cloning is not involved and changes are done on the existing entity itself. `{}` acts as the en-closer.

```
s = $[ a = ?  b = add(a, 1)  c = mul(a, b) ]
s { a = 5  c = sub(a, b) }
print s.a  # 5
print s.c  # -1
```

3.2.5 Translation

Translation takes two entities, one providing the description and the other the implementation. Following are some examples.

```
data = $[1,2,3] @ linked-list
# now entity referred by data has an implementation using a linked list

data = $[1,2,3]
# now entity referred by data has an implementation using the default
# entity implementation provided by the runtime

http-server = $ext @ ext-node("www.iitk.ac.in", 80)
# http-server binds to an entity implemented on
# a server running at www.iitk.ac.in:80
```

3.2.6 Syntactic Extensions

Here, we will discuss features which are extensions to the core syntax of the language. They exist to facilitate writing certain kinds of programs that would otherwise be more complicated.

If-else

This extension is to facilitate writing condition expressions. Its use is self explanatory in the example below. The conditional entity is different from other builtin entities, as there are multiple sources for result, which are selectively activated based on the check condition.

```
x = if less(i,3) then 1 else add(fib(sub(i,1)), fib(sub(i,2)))
# if-else like other entities such as add, sub etc is reactive in nature,
# hence, changes to i, will result in x getting updated accordingly
```

For-each

Sometimes, we need to write multiple statements following a common template. It is cumbersome to write each statement manually in such a case. `forEach` comes to the rescue here. When executed, a `forEach` statement expands the template statement into a collection of statements. See the following example below for a use case.

Example `forEach x in $[1,2,3,4] weHave, <x> = mul(<add(x,1)>, 2)`

is equivalent to

`<1> = mul(<2>, 2)`

`<2> = mul(<3>, 2)`

```
<3> = mul(<4>, 2)

<4> = mul(<5>, 2)
```

When

`when` guards a set of statements, with a condition, such that statements are generated only if the condition evaluates to true.

```
x = 5
when less(x,10) weHave { y = mul(x,2) z = add(x,2) }
# is equivalent to
x = 5
y = mul(x,2)
z = add(x,2)
```

3.3 Implementation

We have implemented an interpreter for JackFruit in Haskell.

Parsing : The Parsec library [8] was used to create a parser for the language.

Default Graph Implementation : The implementation uses a single boxed data structure to represent all kinds of entities.

Builtins : Entities such as add, sub, mul, if-else etc are reactive in nature and are provided as buitins by the interpreter.

Cloning : Each entity keeps track of the sequence of operations required to construct it. We utilize this information to create new copies of existing entities when implementing cloning operation. In case, the existing entity is in non-quiescent state, we defer the cloning to when it achieves quiescent state.

Edges : All the edges have been implemented in the runtime itself as STM transactions. This also ensures the atomicity requirement on computations done during the active phase of an edge (see [9] for proof). Every entity keeps track of edges bound to it. These edges are selectively activated/removed based on changes in the entity and the type of edge.

Association : Every entity, keeps a list of edges bound to it. We modify that list to add new associations.

Specification : Implementing a specification again requires us to adhere to the atomicity requirement, which we achieve using transactions on the top of STM.

Translation : To show translation in action, we have provided a basic implementation in the runtime to interface to entities on remote systems.

Parallelism : At any given time, we have a collection of transactions that need to be executed. STM allows us to execute these transactions in parallel [9]. It automatically manages restarting a transaction, if updates done by it clash with some other transaction. Execution of these transactions results in new associations and specifications and hence more transactions.

Chapter 4

Sample Programs

In this chapter, we will present JackFruit based solutions to a set of problems, encompassing various programming patterns from the combined domain of parallel, concurrent and distributed computations to illustrate the power and flexibility of the language and the underlying CAST framework.

4.1 Pipelining: Sieve of Eratosthenes

The Sieve algorithm is a classical parallel computing algorithm. The task is to generate all primes in the range `[2, n]`. Several variants of the algorithm are possible. We will present a solution based on pipelines. We initially have an input stream generating numbers in the range 2 to n. The algorithm starts by pushing 2 in the output stream and creating a filter which removes numbers divisible by 2 from the input stream and pushes the rest into another immediate stream. We then push the first number in the immediate stream (which must be a prime) in the output stream and create a filter for this prime number. The new filter acts on the immediate stream it receives. We keep on repeating this process and adding filters for each prime till the input is consumed. The parallelism in this solution is obtained by allowing the chain of filters to execute in parallel, with data flowing from one filter to the other similar to a pipeline.

JackFruit implements the pipeline using message passing associations. A specification allows us to create multiple copies of the filter, one for each prime, which can execute in parallel. (refer prog. 4.1)

```
filter = $[ inp = ?
            out = ?
            prime = ?
            x = ?
            x <= inp
            # push number non-divisible by prime into input for next stream
            when not-div-by(prime,x) weHave { x[] => out }
            # repeat for next number in input stream
            # bound check ensures that we have not reached the end of stream
            when bound(x) weHave { filter[inp=‘inp out=‘out prime=‘prime] }
          ]

sieve = $[ inp = ?
           out = ?
           prime = ?
           prime <= inp
           imm-out = $[]
           when bound(prime) weHave
              { prime[] => out
                filter[inp=‘inp, out=imm-out, prime=‘prime]
                sieve[inp=‘imm-out, out=‘out]
              }
         ]

input = count[2,100]
output = $[]
sieve[inp=input, out=output]
```

Program 4.1: Sieve of Eratosthenes

4.2 Fork Join Parallelism : Fibonacci using Recursion

Fork/join parallelism is a style of parallel programming useful for exploiting the parallelism inherent in divide and conquer algorithms. The idea is to divide the larger task into smaller tasks whose solutions can then be combined. Further, as long as smaller tasks are independent, they can be executed in parallel. Recursive version of Fibonacci is one such example, that we present here. (refer prog. 4.2)

```
fib = $[ i = ?
         $result = if less(i, 3)
                      then 1
                      else add(fib(sub(‘i, 1)), fib(sub(‘i, 2)))
       ]

x = fib(7)
```

Program 4.2: Calculating Fibonacci numbers using recursion

4.3 Dataflow Parallelism : Fibonacci using Dynamic Programming

When solving Fibonacci using dynamic programming, we create a dataflow graph representing computational dependencies between various data nodes. The computations are automatically triggered, as soon as its input data becomes available. (refer prog. 4.3)

```
fib = $[ <1> = 1
         <2> = 1
         forEach k in count[3, 7] weHave,
              <k> = add(<sub(k,1)>, <sub(k,2)>)
       ]

x = fib.<7>
```

Program 4.3: Calculating Fibonacci numbers using dynamic programming

4.4 Reactive Programming : Spreadsheets

Reactive programming is an event based programming model, such that computations are fired as soon as input data changes. One of the popular examples of reactive programming is a spreadsheet. JackFruit allows us to convert a simple entity into a spreadsheet by use of change dependency associations. Further, atomicity of specification operations and association computations ensure that the spreadsheet remains consistent under concurrent operations. (refer prog. 4.4)

```
# spreadsheet as an empty entity
s = $[]

# updates to spreadsheet
s { <1> = ?
    <2> = add(<1>, 2)
    <3> = mul(<2>, <1>)
  }

# new updates to spreadsheet
# change dependency ensures that cells are automatically updated
s { <1> = 5
    <3> = add(<2>, <1>)
  }

#spreadsheet now contains <1>=5, <2>=7, <3>=12
```

Program 4.4: Spreadsheet programming

4.5 Concurrent stateful modifications

In this example, we concurrently modify the values referred by name entities using a combination of specification and synchronized assignment, while maintaining consistency. (refer prog. 4.5)

```
state = $[ x := 150
           y := 100
           z := 50
         ]

# if we execute the two transactions below concurrently
state { x := sub(x, 100)   y := add(y, 100) }
state { y := sub(y, 50)   z := add(z, 50) }

# after completion of the above transactions, the state will look like this
# [ x := 50   y := 150  z := 100 ]
```

Program 4.5: Concurrent stateful modifications

4.6 Property Based Parallelism : Addition Reduction

Reduction is a generic operation which takes a binary operation and applies it on a collection of operands. While some binary operations such as addition and multiplication are associative, others such as subtraction and exponentiation are not. Associativity allows reduction to divide the input into smaller inputs, compute them in parallel and finally combine them into the final output. For example, $1 + 2 + 3 + 4$ can be found by calculating $1 + 2$ and $3 + 4$ in parallel, then combining their results to obtain 10 as the result. Such a method, will not work with $1^{2^{3^4}}$, which needs to be calculated sequentially from right to left. Program 4.6 shows how, property entities can help us in exploiting the available parallelism.

```
data = $[1, 2, 3, 4, 5, 6, 7, 8]
sum1 = reduce(add[:associative], 0, data)   # parallelization here
sum2 = reduce(add, 0, data)   # sequential execution here
```

Program 4.6: Addition Reduction

4.7 Data Parallelism : Matrix Multiplication

Data parallelism is effective when we need to perform the same set of operations on different pieces of the data. Then, we can distribute the data among multiple parallel processes. Matrix Multiplication is one such problem, that we present here. (refer prog. 4.7)

4.8 Distributed Computation : MapReduce

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. The major part of MapReduce is to distribute the data to a collection of compute servers and then collect back the results. JackFruit, through its translation operation, allows seamless integration of processes running on compute servers, with other entities on the local machine. (refer prog. 4.8)

```
A = $[ <1,1> = 1, <1,2> = 1
       <2,1> = 1, <2,2> = 1
     ]

B = $[ <1,1> = 1, <1,2> = 1
       <2,1> = 1, <2,2> = 1
     ]

buffer = $[ <1,1> = $[], <1,2> = $[]
            <2,1> = $[], <2,2> = $[]
          ]

forEach i in count[1, 2] weHave,
   forEach j in count[1, 2] weHave,
       forEach k in count[1, 2] weHave,
            mul(A.<i,k>, B.<k,j>) => buffer.<i,j>

C = $[]

forEach i in count[1, 2] weHave,
   forEach j in count[1, 2] weHave,
      C { <i,j> = reduce(add, 0, buffer.<i,j>) }
```

Program 4.7: Matrix Multiplication

```
data = count[1, 6]
mapped-data = ?

# map original data into three parts
mapped-data = map(data, 3)

# initialize the compute nodes
adders = $[ <1> = $ext @ ext-node["127.0.0.1", 4242]
            <2> = $ext @ ext-node["127.0.0.1", 4243]
            <3> = $ext @ ext-node["127.0.0.1", 4244]
          ]

result-data = $[]

# send data to compute nodes
forEach i in $[1,2,3] weHave
   when bound(mapped-data) weHave
      { adders.<i>{ $input = mapped-data.<i> } }

# receive back the results
forEach i in $[1,2,3] weHave adders.<i>.$result => result-data

# reduce to final answer
result = reduce(add, 0, result-data)
```

Program 4.8: A MapReduce program with 3 compute nodes

Chapter 5

Conclusions and Future Work

5.1 Future Work

5.1.1 Complete Implementation of CAST Framework

The JackFruit language and its implementation are works in progress. The current implementation provided by the runtime is a boxed data structure representing all entities. In addition, it involves the overhead of STM for every computation. It does not allow the programmer to add new associations and implementation entities. Instead they are provided by the runtime directly. To provide this ability to the programmer, we will need a collection of basic entities, which can be used to define other entities. One possible way it can be achieved is by creating an interface to the C language. C provides an efficient interface to underlying hardware and the operating system. This will allow us to implement the starting set of basic entities directly in C. Also, there is a scope of further improvement in the language syntax to make it more cleaner and simpler.

5.1.2 Support for Pattern Matching

Pattern matching allows us to express our logic in a succinct manner, by removing the list of conditionals, that we would have to write otherwise. It also eases the implementation of domain specific languages, which can be written as patterns. Current solution for pattern matching include, regular expressions (Awk, Perl), tree patterns (Haskell, ML) and as first class data types in SNOBOL [10]. SNOBOL4 patterns subsume BNF grammars, which are equivalent to context-free grammars and are more powerful than regular expressions. Graphs can provide a good way to express patterns. In JackFruit, we can envision a pattern match as an association between two graphs, one of which represents a pattern. However, exact syntax and semantics for this association needs to be worked out.

5.2 Conclusions

We have presented a framework using a graph abstraction and four operations. The framework successfully subsumes many of the existing abstractions, while retaining its simplicity and succinctness. Further, it does not impose a particular implementation on the user, allowing seamless switching to improve efficiency and performance. A programmer is free to choose abstractions of his/her choice while writing a solution to a problem. It is worth doing a full fledged, efficient implementation of the CAST framework.

References

[1] Concurrency in Erlang & Scala: The Actor Model - Ruben Vermeersch, `http://savanne.be/articles/concurrency-in-erlang-scala`

[2] Why Transactors?, akka documentation, `http://doc.akka.io/docs/akka/2.0/scala/transactors.html`

[3] Circle-ellipse problem, `http://en.wikipedia.org/wiki/Circle-ellipse_problem`

[4] Imperative functional programming - Simon Peyton Jones, Philip Wadler, POPL 1993

[5] On Lisp - Paul Graham, `http://paulgraham.com/onlisptext.html`

[6] Homoiconic languages, `https://blogs.oracle.com/blue/entry/homoiconic_languages`

[7] Gradual Typing for Functional Languages - Jeremy G. Siek, Walid Taha, 2006

[8] Parsec - monadic parser combinator library for Haskell, `http://www.haskell.org/haskellwiki/Parsec`

[9] Software Transactional Memory - Nir Shavit, Dan Touitou, PODC 1995

[10] Pattern Matching, `http://en.wikipedia.org/wiki/Pattern_matching`

[11] ARK - a language for dynamic streaming DAGs, Abhishek Rajput, MTech Thesis, IIT Kanpur, 2011