CPL: A LANGUAGE FOR PROGRAMMING CHART PATTERNS

SASWAT ANAND

(B.Eng., Regional Engineering College,

Rourkela, India)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2002

To My Parents

Acknowledgements

I express my sincere gratitude to *Him* for everything.

I extend my sincere thanks to my advisor A/P Khoo Siau Cheng, working with whom was a very pleasing experience, both personally and intellectually. I appreciate his help and support which were on many occasions, unexpected but certainly very welcome- like his active participation in writing papers all night in the lab. It will always remain amusing to remember how small jokes broke into laughing fits during some of our meetings and were stopped midway.

I would like to thank to A/P Chin Wei Ngan for his valuable and critical comments that helped to improve the quality of our papers very much. I also thank him for supporting me financially.

I appreciate the members of Programming Languages and Systems lab, my friends and my elder brother for giving me good company in Singapore. I want to thank David Chew, whose latex style files for NUS thesis made lexing a breeze.

Last, but not the least I thank my fiancẽe Aliva very much for her understanding and support. I wish I would get an equally helpful and enjoyable company during my PhD.

Contents

Summary

It is an established notion among financial analysts that price moves in patterns and these patterns can be used to forecast future price. As the definitions of these patterns are often subjective, every analyst has a need to define and search meaningful patterns from historical time series quickly and efficiently. However, such discovery process can be extremely laborious and technically challenging in the absence of a high level pattern definition language. In this thesis, we propose a chart-pattern language (*CPL* for short) to facilitate pattern discovery process. Our language enables financial analysts to (1) define patterns with subjective criteria, through introduction of fuzzy constraints, and (2) incrementally compose complex patterns from simpler patterns. We demonstrate through an array of examples how real life patterns can be expressed in CPL. In short, CPL provides a high-level platform upon which analysts can define and search patterns easily and without any programming expertise.

CPL is embedded in Haskell, a state-of-the-art functional language. We show how various features of Haskell, such as pattern matching, higher-order functions, lazy evaluation, facilitate pattern definitions and implementation. Furthermore, Haskell's type system frees the programmers from annotating the programs with types.

Functional language has widely been used as a paradigm for constructing embedded domain-specific applications. Its popularity arises from its ability to (1) support concise syntax for specifying solutions in the domain-specific application, and (2) execute specifications in *interpretive* environment. However, domain-specific languages embedded within functional language such as Haskell usually suffer from inefficient interpretation. In the second part of the thesis, we demonstrate how various features in Haskell, such as laziness and advanced type system, can be exploited to provide an efficient interpretation of our chart-pattern language. CPL enables financial analysts to specify chart patterns through composition of line patterns and constraints. Pattern instances that "match" a chart pattern are then searched and retrieved from the price histories. The search for concrete patterns usually involves constraint solving. We use type class systems with local quantification type annotation to achieve on-the-fly optimization of pattern definition for efficient constraint solving. We also make full use of lazy evaluation to obtain a new and efficient backtracking algorithm for constraint solving.

In the third part of the thesis, we present a novel constraint solving algorithm that the CPL implementation uses for pattern matching. The algorithm is based on divide and conquer framework; it divides the constraint satisfaction problem into smaller subproblems, hence alleviating the curse of dimension and solves each problem independently; the sub-solutions are later combined to form the solution of the original problem. One of the major hurdle in this approach is the inability to divide the problem into smaller parts which are independent, a condition that must be satisfied in order to avoid doing redundant computation for the subproblems. We show how lazy evaluation can be used to overcome this problem. We use this algorithm to solve N-queen problem and compare our result with other algorithms.

List of Tables

Chapter 1

Introduction

"A picture is worth a thousand numbers" when we talk about stock market. Financial analyst is inundated with innumerable sources of information when faced with the challenge to forecast future movement of price. There have been many attempts among both researchers and practitioners to crunch these numbers to forecast more accurately. But for centuries, traders have relied on something other than numbers that tell much more: pictures of price movement(price chart). On price charts, patterns occur and reoccur and every time they bode of similar future events, because these patterns are manifestation of the eternal struggle between bulls and bears. A strong school of market analysts known as *technical analysts* [6, 1] assert that patterns can be used to forecast future price movement. For example, the famous head-and-shoulder is a precursor of substantial decline of price. Therefore, the goal of an investor is to discover meaningful patterns that forecast ensuing price movement from historical market data. But doing so can be a daunting task because of following three reasons:

1. The size of the historical data which the analysts want to search is huge. It is typically of the order of 3000 years of data(300 companies for 10 years).

2. Definitions and interpretations of patterns are very much subjective and reflect individual investor's investment personality. Study of patterns has always been considered an art, rather than a science. Thus, a software with hard-coded definitions for patterns does not cater to the need of all investors.

3. This process of discovering patterns is an iterative process. For example, let us say a user wants to search for head-and-shoulder patterns in the history. To begin the process, he starts with a rough definition of the pattern, and the system fetches him all instances of head-and-shoulder from history that satisfy his definition. But that result set may contain some instances of head-and-shoulder which are not followed by a decline in price. The user can then refine his definition to filter out those undesirable instances. This process of refining definition and pattern mining may continue iteratively.

In this thesis, we propose a high-level language to facilitate pattern discovery process. The language enables financial analysts to do the following tasks:

1. Define patterns with fuzzy constraints. Through incremental addition of fuzzy constraints to a pattern, the user is able to refine patterns iteratively.

2. Reuse patterns. Complex patterns are built by composing simpler patterns and adding further constraints on them.

We call our language *Chart Pattern Language*(CPL). By embedding this language within Haskell, the user can reap benefits from various nice features of Haskell. Specifically,

- Haskell's strong type system infers the type of pattern definition automatically. This frees programmers, who are financial analysts by profession, from the mundane task of declaring variables and specifying types – a task which they are not at all comfortable with or enjoy doing.

- Higher-order functions are used extensively throughout the system to create a natural and concise syntax for the language. CPL has been designed with the aim of making it similar to spread-sheet formula language, which all financial analysts feel familiar with.

- Searching for patterns in price history involves multiple constraints satisfaction, which has a worst case exponential running time. Lazy Evaluation plays a crucial role by automatically avoiding unnecessary computation during searching for patterns.

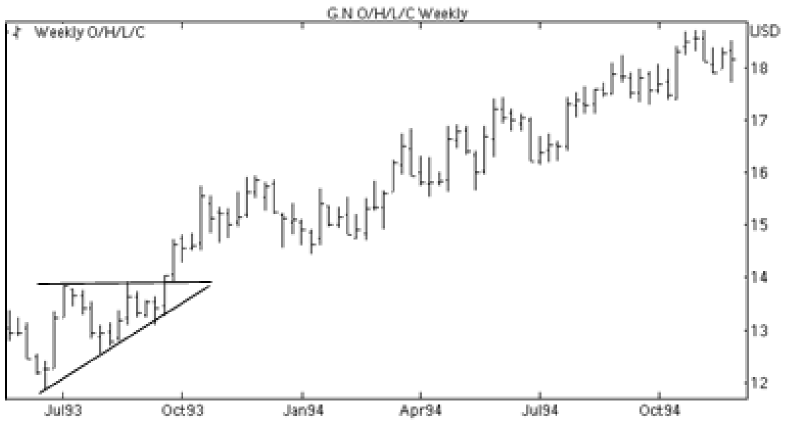

Figure 1.1: Triangle

1.1 What is in a Chart Pattern?

Imagine you are the manager of a mutual fund. Over the years you have purchased a few hundred thousand shares of a company. After seeing the stock rise for almost a year (Figure 1.1), you are getting nervous about continuing to hold the stock. So you tell your trading department to dump all your shares as long as it gets 13.75. Since your fund has a large block of share to get rid of, price cannot rise above 13.75. Word gets around and other big players jump in and start selling. This aggressive selling satiates the demand and price goes down. Eager buyers, viewing the price of the stock as a steal, demand more shares and price starts rising. But when the price touches 13.75 level, your fund sells more shares, effectively halting its advance. The share keeps wriggling up and down a few times and forms a triangle shape-like pattern on the chart, as shown in Figure 1.1. When you are done with selling and supply depletes, buying pressure catapults price above 13.75 level and strong bullish trend is established. So if a chart analyst detects this triangle pattern, he knows what is cooking in the pit, and may wish to get into buying spree once price breaks out from the pattern. This story tells how patterns are formed on charts under particular circumstances and hence it is logical to believe that price can repeat its feat. Some examples of popular patterns are *head-and-shoulder*, *cup-with-handle*, *rectangle*, *triangle*, *flag*, *wedge*, *bump-and-run-reversal*, *etc.*.

Chart patterns define the movement of raw price like closing or opening price or their derivatives. For example, moving average is a running average of price. We call these raw prices and their derivatives *technical indicators*, because they indicate the state of the market. Indicators like moving average are mathematical formulae expressed in terms of the prices/volume associated with a discrete set of time periods.

1.2 Implementing CPL

Functional language has widely been used as a paradigm for constructing domain-specific application. In this paradigm, most of the work required to design, implement and document a domain-specific language (DSL) is inherited from a functional language, such as Haskell [28]. As mentioned by Hudak in his position paper [18], ‘DSL [in Haskell] can be viewed as a higher-order algebraic structure, a first-class value that has the “look and feel” of syntax.’ Furthermore, the interpretive nature of Haskell, and functional languages in general, provides a friendly and effective framework for the usually small programs in DSL to be executed.

However, running DSL programs in interpretive mode can be slow, because it precludes any opportunity to optimize the program. While domain-specific compilation [10, 21] attempts to solve this efficiency problem, it does so by reducing the interpretability of the embedded language; the resulting language is no more an embedded language in true spirit. Eventually, it limits the profitability of embedded approach for rapid prototyping.

In Chapter 4 we address the issues of inefficient interpretation of CPL. CPL models the domain-specific entities like patterns and technical indicators as functions. Functions which in one hand is a natural and intuitive choice, they are not amenable to optimizations that can be done if patterns were modeled as data types.

We address this problem of optimization by exploiting Haskell's *type class system*; we overload some functions in our language with new data type. Using this new type, we can change the behavior of a function into one that reveals its own internal structure at interpretation time. This internal structure enables the interpreter to avoid redundant computation. Furthermore, we utilize local quantification annotation feature of Haskell to overcome the typing problem that arises from this overloading.

1.3 Lazy Divide & Conquer Constraint Solving

The pattern definitions of CPL are constraint based. Searching for patterns is thus a Constraint Satisfaction Problem(CSP). CPL uses an innovative constraint solving algorithm called "lazy divide & conquer".

Constraint Satisfaction Problems (CSP) have drawn a great deal of attention because of their simple and general definitions and wide application in diverse fields. For the past two decades, there have been numerous improvements of the naive backtracking algorithm, which was proposed a century before. Some of the popular ones are, backjumping, backmarking, forward checking, arc consistency, and of course their various combinations and variants.

However, none of these algorithms has adopted divide and conquer (**d&c**) paradigm. Although there are schemes for identifying constraint networks that are decomposable [8, 15], these works restrict the manner in which a problem can be decomposed.

The main reason for *not* expressing constraint solving algorithms in d&c paradigm is the difficulty in cleanly dividing a constraint satisfaction problem into independent subproblems, with no constraint inter-connecting these subproblems. If global constraints that inter-connect subproblems exist, a naive solution, in the spirit of d&c, is to temporarily ignore these constraints, solve the subproblems independently, and later check for consistency of those global constraints. This method might cause many unnecessary consistency checks during solving of subproblems. Specifically, the existence of global constraints inter-connecting two subproblems might make their solutions incompatible.

On the other hand, such "over-checking" of constraints on a subproblem can be eliminated by working on the subproblems *only on demand*. Programming-wise, this amount to solving problems under *lazy evaluation*.

Lazy evaluation is a notion that comes from functional programming language like Haskell [28], where a computation is performed if and only if its result is required for further computations. For example, consider the following definition of a function `f` with parameter `x` and function body that always returns 2.

```
fun f x = 2
```

When the function `f` is called, its actual argument – which might be a complicated expression – is never evaluated because it is not needed for calculating the result of `f` (which is `2`.)

In Chapter 5 we describe a lazy d&c strategy for constraint solving, and show that this strategy can significantly reduce the number of consistency checks. We apply this algorithm to finding CPL patterns and N-queen problem.

Chapter 2

Technical Indicators

Stock charts are usually displayed as *bar* charts. Here, there is one bar for each time period. One time period can be a day on a daily chart, or a week on a weekly chart, *etc.*. Each bar is associated with five kinds of values for that time period; they are: the close price, open price, high price, low price, and the number of transactions (volume) for that time period. Throughout the thesis, we use bar, time and "a day" interchangeably without loss of generality.

We represent bars by integers, and technical indicators are hence functions from bars to values.

$$\textbf{type}\ \mathit{Indicator}\ a\ =\ \mathit{Bar}\ \rightarrow\ (\mathit{Maybe}\ a)$$

Indicator takes a bar b and returns an indicator value for that bar. Since each bar is associated with five basic fields: high, low, open, close price, and transaction volume, we have five primitive indicators: *high*, *low*, *open*, *close* and *volume*. Thus, we have

$$\mathit{high},\ \mathit{low},\ \mathit{open},\ \mathit{close}\ ::\ \mathit{Indicator\ Price}$$
$$\mathit{volume}\ ::\ \mathit{Indicator\ Volume}$$

More complicated indicators can be composed of the simpler ones, with the help of a combinator library.

As the first example, we define a technical indicator for *typical price*, which is the average of the high, low and close prices of a day:

$$\begin{aligned}&\mathit{tPrice} :: \mathit{Indicator}\ \mathit{Price}\\&\mathit{tPrice} = (\mathit{high} + \mathit{low} + \mathit{close})/3.0\end{aligned}$$

Note that any arithmetic combination of indicators is also an indicator, because indicators are an instance of the *Num* class, which has operations for addition, subtraction, multiplication, *etc.*:[1] Indicators and their operations are reminiscent of Frans' *behaviours* [11], and of *observable* used in composing financial contracts [27]; operations on them are supported by combinators like *lift*1, *lift*2, *etc.* that lift functions to the indicator level, as follows,

$$\begin{aligned}&\mathit{lift}1' :: (a \rightarrow b) \rightarrow (\mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ b)\\&\mathit{lift}1'\, f\ \mathit{Nothing} = \mathit{Nothing}\\&\mathit{lift}1'\, f\ (\mathit{Just}\ a) = \mathit{Just}\ (f\ a)\\[1em]&\mathit{lfit}2' :: (a \rightarrow b \rightarrow c) \rightarrow (\mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ b \rightarrow \mathit{Maybe}\ c)\\&\mathit{lift}2'\, f\ (\mathit{Just}\ a)\ (\mathit{Just}\ b) = \mathit{Just}\ (f\ a\ b)\\&\mathit{lift}2'\ \_\ \_\ \_ = \mathit{Nothing}\\[1em]&\mathit{lift}1 :: (a \rightarrow b) \rightarrow (\mathit{Indicator}\ a \rightarrow \mathit{Indicator}\ b)\\&\mathit{lift}1\ f = \lambda\, i\ \mathit{bar}\, .\, (\mathit{lift}1'\, f)\ (i\ \mathit{bar})\\[1em]&\mathit{lift}2 :: (a \rightarrow b \rightarrow c) \rightarrow (\mathit{Indicator}\ a \rightarrow \mathit{Indicator}\ b \rightarrow \mathit{Indicator}\ c)\\&\mathit{lift}2\ f = \lambda\, i1\ i2\ \mathit{bar}\, .\, (\mathit{lift}2'\, f)\ (i1\ \mathit{bar})\ (i2\ \mathit{bar})\end{aligned}$$

[1]It suffices to understand that the usual arithmetic operations work on indicators too.

```
instance (Num a) ⇒ Num (Maybe a) where
  fromInteger = Just . fromInteger
  abs = lift1′ abs
  (+) = lift2′ (+)
  ...
instance (Num (Maybe a)) ⇒ Num (Indicator a) where
  fromInteger = const . fromInteger
  abs = lift1 abs
  (+) = lift2 (+)
  ...
```

const is a Haskell prelude function that always returns its first argument. Its type is, $a \rightarrow b \rightarrow a$.

It is very common while defining indicators to use past values of an indicator. To support this, we have a combinator ($\sharp$) which enables *relative* indexing into past data. Given a bar (time), while an indicator, say *high*, evaluates to the high price at the bar, *high* $\sharp$ n yields the high price of nth previous bar. ($\sharp$) is defined as follows:[2]

```
(♯) :: Indicator a → Integer → Indicator a
ind ♯ n = λ b . ind (b − n)
```

Relative indexing can used to define *Moving average*. This is a popular indicator which represents current market trend by smoothing small price fluctuations. An "n-day" moving average of an indicator on a given day is calculated by averaging the values of the indicator for previous n days:

```
movingAvg :: (Num a) ⇒ Integer → Indicator a → Indicator a
movingAvg n price = (sum lastnbar )/n
```

[2]In Haskell, an infix operator can be turned into a function by enclosing it in braces. Conversely, a function can be turned into an infix operator by enclosing it in backquotes.

$$\textbf{where} \quad \mathit{lastnbar} = \mathit{takeI}\ n\ \mathit{price}$$

Function *sum* is a standard Haskell function for summing up a list. *takeI* retrieves the value of an indicator for the last n days, and is defined through relative indexing:

$$\mathit{takeI} :: \mathit{Integer} \rightarrow \mathit{Indicator}\ a \rightarrow [\mathit{Indicator}\ a]$$
$$\mathit{takeI}\ n\ f = \mathit{map}\ (f\ \sharp)\ [0\ ..\ (n-1)]$$

Day	0	1	2	3	4	5	6	7
Close Price	4.2	4.5	4.3	4.1	4.5	4.6	4.3	4.4
Day	8	9	10	11	12	13	14	15
Close Price	4.2	4.3	4.4	4.1	3.9	3.6	3.7	3.8

In the above table if we evaluate (*takeI* 5 *close*) 15, we get a list of closing prices from 15^{th} day down to 11^{th} day.

Now we can explain why an indicator evaluates to a value of the *Maybe* type. *Maybe* is defined in Haskell as follows [28]:

$$\mathit{data\ Maybe\ a} = \mathit{Nothing} \mid \mathit{Just}\ a$$

From the "n-day" moving average definition we can see that the first bar in the price history when moving average can be calculated is the nth bar. Before this bar, computation of "n-day" moving average will require prices before the start of history, which is not available. By making the result of indicator *Maybe* type, successful computation of an indicator will return a value of the form *Just* v, where v is the desired value. Value *Nothing* denotes an error in computation, as explained above.

One of the popular use of moving average is to compare it with the raw price, say close price. A buy signal is generated when the security's current price rises from below its moving average to above it, and a sell signal is generated if current price falls from above its moving average to below it. Following are definitions of the above rule in CPL:

$$
\begin{aligned}
&\textit{buy},\ \textit{sell}\ ::\ \textit{Indicator Bool}\\
&\textit{buy}\ = \textit{close}\ \text{`}\textit{riseAbove}\text{`}\ (\textit{movingAvg}\ \textit{close}\ 13)\\
&\textit{sell}\ = \textit{close}\ \text{`}\textit{fallBelow}\text{`}\ (\textit{movingAvg}\ \textit{close}\ 13)\\
&\textit{riseAbove},\ \textit{fallBelow}\ ::\ (\textit{Num}\ a)\ \Rightarrow\\
&\qquad \textit{Indicator}\ a\ \rightarrow\ \textit{Indicator}\ a\ \rightarrow\ \textit{Indicator Bool}\\
&f1\ \text{`}\textit{riseAbove}\text{`}\ f2\ = (f1\ >\ f2)\ \&\&\ (f1 \sharp 1)\ <\ (f2 \sharp 1)\\
&f1\ \text{`}\textit{fallBelow}\text{`}\ f2\ = (f1\ <\ f2)\ \&\&\ (f1 \sharp 1)\ >\ (f2 \sharp 1)
\end{aligned}
$$

As shown in Figure 2.1, due to declarations 4 and 8 indicators can be compared and logically combined through (&&) and (||) operations as in the above example.

A more complex example of technical indicator is *momentum*. This formula computes a number (between 0 and 1) that signifies the momentum of the last 14-day trend. If the momentum is high (closer to 1), the trend is bullish, and money should be put along the trend. Momentum is calculated in terms of the total price increase in all the "up days" and the total decrease in all the "down days". A day is called an "up day" if its close is greater than its previous day's close; otherwise, it is called a "down day". For example, in the preceding table the up days are 1,4,5,7,9,10,14,15 and the down days are 2,3,6,8,11,12,13. The momentum is calculated by the following formula:

$$\textit{momentum}\ =\ \textit{sum\_up}\ /\ (\textit{sum\_down}\ +\ \textit{sum\_up})$$

where *sum_up* sums up the "diff. of a day" for all up days, and *sum_down* sums up the "diff. of a day" for all down days. A "diff. of a day" is the absolute difference between the day's close and the previous day's close. In CPL this translates to:

$$
\begin{array}{l}
\mathit{momentum} :: \mathit{Integer} \rightarrow \mathit{Indicator\ Price} \\
\mathit{momentum}\ n\ =\ \mathit{sum\_up} \,/\, (\mathit{sum\_down}\ +\ \mathit{sum\_up}) \\
\quad \textbf{where} \\
\quad\ \mathit{sum\_up} \quad\ \ = \mathit{sum}\ (\mathit{takeI}\ \ n\ \mathit{up\_day}) \\
\quad\ \mathit{sum\_down}\ = \mathit{sum}\ (\mathit{takeI}\ \ n\ \mathit{down\_day}) \\
\quad\ \mathit{up\_day} \quad\ \ = \mathit{ifI}\ (\mathit{close}\ >\ \mathit{close} \sharp 1)\ (\mathit{close}\ -\ \mathit{close} \sharp 1)\ 0 \\
\quad\ \mathit{down\_day}\ = \mathit{ifI}\ (\mathit{close}\ \leq\ \mathit{close} \sharp 1)\ (\mathit{close} \sharp 1\ -\ \mathit{close})\ 0
\end{array}
$$

Function *ifI* mimics the conditional operation at the indicator level; it is defined as follows:

$$
\begin{array}{l}
\mathit{ifI}\ :: \mathit{Indicator\ Bool} \rightarrow \mathit{Indicator\ a} \rightarrow \mathit{Indicator\ a} \\
\mathit{ifI}\ \mathit{cond}\, f1\, f2\ =\ \lambda\, b\ .\ \textbf{case}\ (\mathit{cond}\ b)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \begin{array}{ll}
\mathit{Nothing} & \rightarrow \mathit{Nothing} \\
\mathit{Just\ True} & \rightarrow f1\ b \\
\mathit{Just\ False} & \rightarrow f2\ b
\end{array}
\end{array}
$$

To a (non-functional) programmer, the fact that momentum is defined over a period of time may indicate a use of `for`-loop construct. But to many financial analysts, this seemingly innocuous control structure pose a steep learning curve, whereas they are very comfortable with the formula-like language of spreadsheets. A little reflection on how *momentum* is calculated in a spreadsheet application reveals much similarity to the above definition. So, anybody who can work with a spreadsheet will find CPL easy to use. To this effect, higher-order functions play a crucial role in creating such an intuitive spreadsheet-like syntax.

We end this section by introducing a technical indicator which is recursively defined. One of the popular recursive indicators is *exponential moving average*. This variant of moving average is exponentially weighted such that more recent values are given higher weight than older values. The formula for an exponential moving average (for closing price) at time t, $emvg_t$, is as follows:

$$
\begin{aligned}
emvg_t &= \alpha * emvg_{t-1} + (1-\alpha) * close_t,\ 0 < \alpha < 1 \\
emvg_0 &= close_0
\end{aligned}
$$

The exponential moving average for the first day (day 0) in a price history is just its closing price. At any other day, the exponential moving average is computed by summing the weighted closing price of that day and the weighted moving average of the previous day. In CPL, this can be defined as follows:

```
    emvg :: Indicator Price
    emvg = (0.8 * (emvg♯1) + 0.2 * close) `seed` close

    seed :: Indicator a → Indicator a → Indicator a
    seed i1 i2 = λ b . if (r == Nothing) then i2 b
                         else r
      where   r = i1 b
```

In the definition above, α is taken as 0.8. *seed* function takes two indicators $i1$ and $i2$, where $i2$ represents the seed indicator. Given a bar b, *seed* returns the value of seed indicator $i2$ if $(i1\ b)$ cannot be evaluated. Otherwise, it returns the value $(i1\ b)$.

1. **class** *CrispOrd a b* **where**

   $(>), (<), (==) :: a \to a \to b$

2. **instance** $(Ord\ a) \Rightarrow$ *CrispOrd a Bool* **where**

   ...

3. **instance** $(Ord\ a) \Rightarrow$

   CrispOrd (*Maybe a*) (*Maybe Bool*) **where**

   ...

4. **instance** $(Ord\ (Maybe\ a)) \Rightarrow$

   CrispOrd (*Indicator a*) (*Indicator Bool*) **where**

   ...

5. **class** *Logic a* **where**

   $(\&\&), (||) :: a \to a \to a$

6. **instance** *Logic Bool Bool* **where**

   ...

7. **instance** $(Logic\ a) \Rightarrow$ *Logic* (*Maybe Bool*) (*Maybe Bool*) **where**

   ...

8. **instance** $(Logic\ (Maybe\ Bool)) \Rightarrow$

   Logic (*Indicator Bool*) (*Indicator Bool*) **where**

   ...

Figure 2.1: A Partial List of Haskell Type Classes for CPL

Chapter 3

Chart Patterns

Chart patterns are geometrical shapes. They capture general trends or movements of price over a period of time. We adopt a constructive approach for defining chart patterns: chart patterns are built from six primitive patterns – *bar*, *up*, *down*, *horizontal*, *resistance line* and *support line*, their derivatives and composites.

1. The pattern *bar* characterizes price movement at a particular time.

2. The price movement from a time a to a time b constitutes an *up* pattern if $\text{high}_t < \text{high}_b$, for all $t \in [a, b)$ and $\text{low}_t > \text{low}_a$, for all $t \in (a, b]$.

3. The price movement from a time a to b constitutes a *down* pattern if $\text{low}_t > \text{low}_b$, for all $t \in [a, b)$ and $\text{high}_t < \text{high}_a$, for all $t \in (a, b]$.

4. The price movement from a time a to b constitutes a *horizontal* pattern if close_t is *approximately* the same as close_b, for all $t \in [a, b)$.

5. A line segment from a time a to b constitutes a *resistance line* pattern if for all $t \in [a, b]$, high_t falls on or below the line, and there are at least 2 days in (a, b) when the respective high prices falls *on* the line.

6. A line segment from a time a to b constitutes a *support line* if for all $t \in (a, b)$, low$_t$ falls on or above the line, and there are at least 2 days when (a, b) when the respective low prices falls *on* the line.

We distinguish between a pattern and a *pattern instance*. A pattern is a description of a shape and a pattern instance is an occurrence of the pattern in the price history. Type-wise, patterns are of type *Pattern*, and pattern instances are of type *Patt*.

Primitive patterns are defined by the primitive functions *bar*, *up*, *down*, *hor*, *res* and *sprt*, respectively.

3.1 Landmarks & Components

Study in human vision system [7] shows that humans find it natural to describe patterns based on their anchor points. In their proposal of a new model for pattern matching in time-series databases [26], Perng *et al.* call these anchor points *landmarks*, and show that they are crucial for effective pattern matching. We follow the same terminology for the anchor points of a chart pattern/pattern instance. Furthermore, we often use landmarks to define constraints of complex patterns. Landmarks are referred to by their positions in the price history, *i.e.*, the values of type *Bar* in CPL. Thus, an up pattern instance has two landmarks: the start and end points of the instance and so have the other primitive "line" patterns: down, horizontal, resistance and support lines. On the other hand, a bar pattern instance has just one landmark, indicating the time when the bar occurs. We will describe the landmarks of complex pattern instances when we discuss pattern composition. In CPL, we have a built-in function $lms\ ::\ Patt\ \rightarrow [Bar]$, that takes in a pattern instance and returns its list of landmarks.

Besides using landmarks, we can also directly access the sub-components of a composite pattern. $sub\ ::\ Patt\ \rightarrow [Patt]$ returns a list of the component pattern instances of a composite pattern instance. For primitive pattern instances, *sub* returns a singleton list containing the primitive. We shall provide details of *lms* and *sub* in Section 3.3.

3.2 Constrained-by Operation

The first pattern we shall present is a bar pattern called *inside day*. This pattern describes the market behavior of a stock for one day. In Figure 3.1, each inside day has a circular mark above it. Following is a definition of inside day:

$$
\begin{aligned}
&\mathit{insideDay} :: \mathit{Pattern}\\
&\mathit{insideDay} = \mathit{bar} \bowtie \lambda\, u\, .\, \mathbf{let}\, [t] = \mathit{lms}\ u\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\, [\, \mathit{low}\ t > (\mathit{low} \sharp 1)\ t,\\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathit{high}\ t < (\mathit{high} \sharp 1)\ t]
\end{aligned}
$$

The above declaration can be interpreted as follows: An inside day is a bar pattern such that, for any of its pattern instances occurring at a time t, its low price is greater than the previous day's low, and its high price is less than the previous day's high.

The ($\bowtie$) operation associates a *constraint function* to a pattern. It has the following type:

$$(\bowtie) :: (\mathit{ToFuzzy}\ a) \Rightarrow \mathit{Pattern} \rightarrow (\mathit{Patt} \rightarrow [a]) \rightarrow \mathit{Pattern}$$

This operation provides an elegant and high-level way of defining patterns. A pattern can be defined in two parts: in the first part, sub-components are defined. In the second part, a constraint function specifies a list of global constraints about the pattern. These constraints are usually expressed in terms of the pattern's landmarks and sub-components. They will be evaluated to values, the types of which belong to the class *ToFuzzy*.

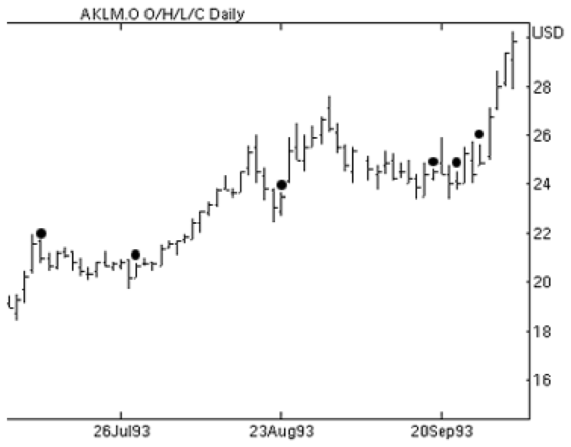

Figure 3.1: Inside days

$$
\begin{array}{l}
\textbf{class } \mathit{ToFuzzy}\ a \textbf{ where} \\
\quad \mathit{toFuzzy} :: a \rightarrow \mathit{Fuzzy}
\end{array}
$$

Basically, a type belongs to the class *ToFuzzy* if its values can be converted to fuzzy values. Currently, these types are: *Fuzzy*, *Bool*, *Maybe Fuzzy*, and *Maybe Bool*.

Notice that the constraint function is defined over a pattern instance, rather than a pattern. This is necessary, as the constraint function is evaluated with respect to a pattern instance. However, it is not wrong to view the constraint function as part of a pattern, since the constraint function applies consistently to *all* instances of that pattern. Similarly, we apply *lms* function to a pattern instance, even though we frequently talk about the landmarks of a pattern.

Now let us introduce fuzzy constraints through a simple pattern called *bigUp*. This is a primitive *up* pattern with a constraint function specifying that increase in price is *usually greater* than 10 units. In CPL, this can be defined as follows:

$$
\begin{array}{l}
\mathit{bigUp} :: \mathit{Pattern} \\
\mathit{bigUp} = \mathit{up} \bowtie \lambda\, u.\, \textbf{let}\ [p, q] = \mathit{lms}\ u \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ [(\mathit{high}\ q - \mathit{low}\ p) \succ 10]
\end{array}
$$

1. **class** *FuzzyOrd a b* **where**

   $(\succ), (\prec), (\asymp) :: a \rightarrow a \rightarrow b$

2. **instance** $(Ord\ a) \Rightarrow$ *FuzzyOrd a Fuzzy* **where**

   ...

3. **instance** $(Ord\ a) \Rightarrow$

   FuzzyOrd (*Maybe a*) (*Maybe Fuzzy*) **where**

   ...

4. **instance** $(Ord\ a) \Rightarrow$

   FuzzyOrd (*Indicator a*) (*Indicator Fuzzy*) **where**

   ...

Figure 3.2: A Partial List of Haskell Type Classes for CPL

Note the use of fuzzy operator $\succ$ in defining the constraint. This reflects some degree of subjectivity in the definition of a pattern, as is frequently done in describing patterns that cannot be quantified exactly. Consequently, it is necessary to attach each pattern instance with a fuzzy value, so as to provide the user a gauge on how closely that instance matches the specified pattern.

Fuzzy values are floating-points between 0.0 and 1.0:

type *Fuzzy* = *Float* — [0.0, 1.0]

For example, $4.9 \succ 5.0$ and $4.9 \asymp 5.0$ both have truth values greater than 0 and less than 1, whereas both $4.9 > 5$ and $4.9 = 5$ have the truth value *False*. Evaluation of fuzzy constraints is done through the support of fuzzy *membership functions* [22]. In the example of $4.9 \asymp 5.0$, we can use the following triangular function to compute the fuzzy value:

$$f\ a\ x\ y\ =\ \begin{cases} (a + x - y)\ \div\ a,\ y\ >\ x \\ (a + y - x)\ \div\ a,\ y\ \leq\ x \end{cases}$$

The function f computes the fuzzy value for $y \asymp x$. The value of a in the function determines the amount of fuzziness. The bigger a's value is, the more values lying farther from x will have non-zero truth value. In CPL, a default value is set for a that can be tuned by experienced users. For further details on coding fuzzy logic in Haskell readers are referred to the work by Meeham and Joy [23].

To support fuzzy operations, We define *FuzzyOrd* class as shown in Figure 3.2. The *FuzzyOrd* class of operators are $\succ$, $\prec$, $\asymp$. *CrispOrd* class, shown in Figure 2.1 consists of conventional crisp operators $>$, $<$ and $==$, which return values of types *Bool*, *MaybeBool* or *Indicator Bool* depending on the type of operands. Similar to crisp operators these fuzzy operators too return *Fuzzy*, *Maybe Fuzzy* and *Indicator Fuzzy* types depending on the type of operands.

3.3 Composing patterns

In this section, we describe how simple patterns can be composed to construct interesting and useful chart patterns. We introduce two operations, "followed-by" and "overlay", which enables the construction of most of the line patterns described in the Encyclopedia of Chart Patterns [6].

3.3.1 Followed-By Composition

We demonstrate followed-by composition by defining a pattern *hill*, which has an *up* pattern followed by a *down* pattern, such that the two "bottoms" of the hill (*i.e.*, the start point of *up* and the end point of *down*) are *almost at the same level*. In CPL, we use the followed-by operation, denoted by ($\gg$), to concatenate these two patterns, and the constrained-by operation ($\bowtie$) to express the global constraint of the *hill*.

$$\begin{aligned} \mathit{hill} \;=\; (\mathit{up} \gg \mathit{down}) \bowtie \lambda\, h.\, & \mathbf{let}\, [a, b, c] \;=\; \mathit{lms}\; h \\ & \mathbf{in}\, [\mathit{low}\; a \asymp \mathit{low}\; c] \end{aligned}$$

The $(\gg)$ operation has the following type:

$$(\gg) :: \mathit{Pattern} \to \mathit{Pattern} \to \mathit{Pattern}$$

It takes in two patterns and concatenates them so that the two sub-patterns join at one point. Specifically, the rightmost landmark of the left pattern-operand must be identical to the leftmost landmark of the right pattern-operand. In listing its landmarks, we list the landmarks of its left operand, followed by the landmarks of its right operand, while ensuring that the common landmark is not duplicated. Consequently, the number of landmarks of the composite pattern will be one less than the sum of the number of landmarks of the two sub-patterns. A *hill* pattern instance thus has three landmarks: the left hill bottom, the peak, and the right hill bottom, in that order.

We mentioned briefly before that function *sub* retrieves sub-components of a pattern instance. For a pattern composed via followed-by operation, *sub* operates recursively on its components to retrieve a list of component-pattern instances which are either primitive or overlay pattern instances. (An *overlay pattern* is a composite pattern whose top-level composition is an overlay operation.) In this list, components of left operand are placed before those of right operand. For example, applying *sub* to a *hill* pattern instance will yield a list $[p, q]$, where p and q are the *up* instance and the *down* instance respectively.

We now use the *hill* pattern to further compose a pattern that can actually earn you big money. *Head-and-Shoulder* is one of the most popular patterns. It looks like as its name suggests. As Figure 3.3 shows, we can define this pattern as a back to back concatenation of three hills, such that the middle hill (head) is higher than the other two (shoulders), and the peaks of the shoulders are *almost* at the same height. In CPL, this translates to,

$$\begin{aligned}
\textit{head\_shoulder} \;=\; & (\textit{hill} \gg \textit{hill} \gg \textit{hill}) \bowtie \lambda\, hs\, . \\
& \quad \textbf{let}\, [a, b, c, d, e, f, g] \;=\; \textit{lms hs} \\
& \quad \textbf{in}\, [\, \textit{high}\ d \;>\; \textit{high}\ b, \\
& \qquad \textit{high}\ d \;>\; \textit{high}\ f, \\
& \qquad \textit{high}\ b \;\asymp\; \textit{high}\ f\,]
\end{aligned}$$

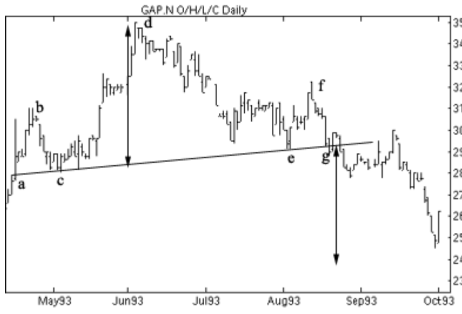

Figure 3.3: Head and Shoulder

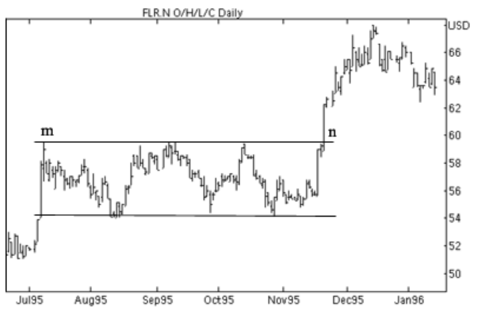

Figure 3.4: Rectangle

3.3.2 Overlay Composition

It is not uncommon to see multiple patterns appearing during a single time interval of the price history, with each pattern measuring different aspects of price movement. In CPL, this is accomplished by the *overlay* operation, denoted by $(<>)$.

$$(<>) \;::\; \textit{Pattern} \;\rightarrow\; \textit{Pattern} \;\rightarrow\; \textit{Pattern}$$

This takes in two patterns, and produces a composite pattern, where the two input patterns occur exactly at the same time period. Specifically, both patterns have the same leftmost landmark, as well as the same rightmost landmark. For an overlay pattern instance, *lms* function returns the landmarks of its left operand, followed by the landmarks of its right operand, while ensuring that the common landmarks are not duplicated. Furthermore, the first and last landmarks in the result list is the first and last landmarks of any of its operand respectively. Consequently, the number of landmarks of the composite pattern will be two less than the sum of the number of landmarks of the component patterns.

Applying *sub* function to an overlay pattern returns components that are either primitive or followed-by instances. (A *followed-by instance* is a composite instance whose top-level composition is the followed-by operation.) Like in case of followed-by operation, this list of pattern instances are then ordered such that components of the left operand of the overlay operator are listed before the components of the right operand.

As an example of overlay operation, we define an *area* pattern which is obtained by overlaying a support-line pattern on a resistance-line pattern. The resulting pattern is so simple that by its own, it is not useful, but it can be used as a building block for other meaningful patterns.

$$area \;=\; res \;<>\; sprt$$

This *area* pattern has two landmarks, signifying the starting and the ending time of the pattern (these are also the starting and the ending time of the components: *res* and *sprt*).

Now, consider an *area* pattern that ends at a time when the closing price either crosses above the resistance line or falls below the support line. Price breaking out of any of the two lines is known as “breakout”. An *upArea* pattern is an *area* with an upside break out; *i.e.*, close price moves past the resistance line. Dually, a *downArea* pattern is an *area* with downside breakout. In CPL, they are defined as follows:

$$\begin{aligned} downArea \;=\; area \bowtie \lambda\, u\, .\, \mathbf{let}\, [m,n] \;&=\; sub\;\; u \\ [a,b] &= lms\;\; n \\ \mathbf{in}\, [breakout\;\; n\;\; b]& \end{aligned}$$

In these examples, m and n returned from evaluating $(sub\;\; u)$ are the instances of resistance line and support line respectively. Function *breakout* takes in a line(either a support or a resistance line) and a landmark b. If the input line is a support line (n), it evaluates to *True* in case the price goes below the support line at time $(b+1)$. Otherwise, it evaluates to *False*. If the input line is a resistance line (m), then it evaluates to *True* in case the price goes above the line at time $(b+1)$. Else it returns *False*.

With these building blocks, we can define a useful pattern called *rectangle*. This pattern denotes confusion and uncertainty of the market. It is developed when there is doubt about the price of an underlying stock; *i.e.*, traders do not know if the stock should be priced higher or lower, and the stock's price keeps fluctuating in a very narrow range which is defined by two *nearly* parallel lines. When a rectangle pattern occurs, financial analysts look for the price to break in either direction. Following is a definition of *rectangle* pattern constructed on top of *downArea*.

$$\begin{aligned} rect \;=\; downArea \bowtie \lambda\, u\, . & \\ & \mathbf{let}\, [m,n] \;=\; sub\;\; u \\ & \mathbf{in}\, [slope\;\;\; m \asymp slope\;\;\; n] \end{aligned}$$

Function *slope* takes either a support or a resistance line and returns its slope.

Our next example is an exotic pattern called *diamond*. Diamond is formed by overlaying a series of support lines on a series of resistance lines. Figure 3.5 shows a diamond on chart. This pattern generally occurs before price starts trending in either upward or downward direction, depending on whether the price breaks up or down from the diamond respectively. In CPL, a downward breaking diamond is defined as follows:

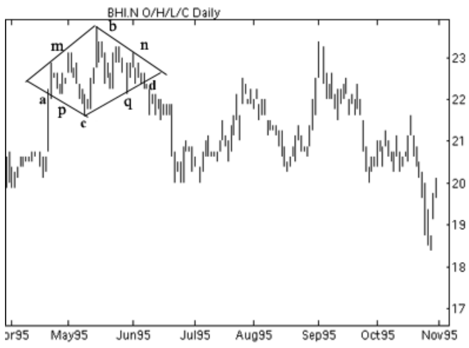

Figure 3.5: Diamond

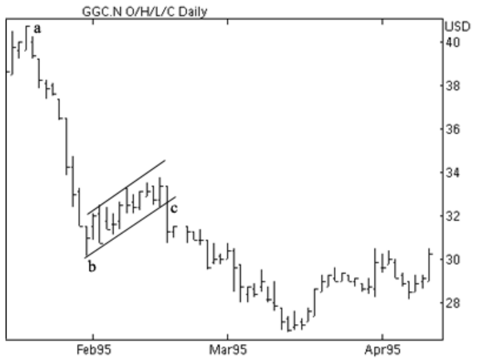

Figure 3.6: Flag

$$
\begin{aligned}
&\mathit{dmnd} \;=\; ((\mathit{res} \gg \mathit{res}) <> (\mathit{sprt} \gg \mathit{sprt})) \bowtie \\
&\quad \lambda\, u\,.\,\textbf{let}\; [\mathit{rlines}, \mathit{slines}] = \mathit{sub}\; u \\
&\qquad\qquad [m, n] = \mathit{sub}\; \mathit{rlines} \\
&\qquad\qquad [p, q] = \mathit{sub}\; \mathit{slines} \\
&\qquad\qquad [c, d] = \mathit{lms}\; q \\
&\qquad\; \textbf{in}\; [\mathit{diverge}\;\; m\; p,\; \mathit{converge}\;\; n\; q,\; \mathit{breakout}\;\; q\; d\,]
\end{aligned}
$$

Given a diamond instance u, $(\mathit{sub}\; u)$ returns its components, the resistance lines (rlines) and the support lines (slines) respectively. These lines can be further decomposed using the *sub* function. Graphically, the components are labeled as follows:

$$
\overbrace{(\underbrace{\mathit{res}}_{m} \gg \underbrace{\mathit{res}}_{n})}^{\mathit{rlines}} <> \overbrace{(\underbrace{\mathit{sprt}}_{p} \gg \underbrace{\mathit{sprt}}_{q})}^{\mathit{slines}}
$$

In the constraint function specification, both *converge* and *diverge* functions take two lines(either support or resistance) and return *True* if these two lines converge or diverge respectively, otherwise they return *False*.

We end the section by defining a pattern constructed via both composition operations. A *flag* pattern is formed when a rapid steep decline of the price is followed by a congestion area where the price moves between two parallel lines (a rectangle) for some days before falling off in downward direction. An instance of a flag pattern is shown in Figure 3.6. A financial analyst may wish to define a flag as follows:

$$
\begin{aligned}
\mathit{flag} \;=\; (\mathit{down} \gg \mathit{rect}) \bowtie \lambda f\,.\;
& \textbf{let}\, [dn, rt] = \mathit{sub}\ f \\
& \quad\; [a, b, c] = \mathit{lms}\, f \\
& \quad\; pr \qquad\;\; = \mathit{high}\ a \\
& \textbf{in}\, [(\mathit{len}\ dn) \prec 5, \\
& \quad\; pr - (\mathit{low}\ b) \succ 0.1 * pr, \\
& \quad\; (\mathit{len}\ rt) \prec 10]
\end{aligned}
$$

Here, a flag is defined by an *Down* pattern followed by a *rectangle* pattern(defined earlier). The landmarks are: a marks the beginning of *down*; b denotes the joint between *down* and *rect* as well as the start-point of *rect*; c the end-point of *rect*. Component-wise, *dn* refers to the *down* pattern, *rt* refer to the rectangle. The constraint function associated with the flag specifies that the *down* pattern *usually* lasts less than 5 days, the price drop during *down* pattern *is around* 10% or more, and price congestion (the *rect*) should *usually* last less than 10 days. *len* function returns the difference of the end and the start landmarks of a pattern.

In summary, we have described an effective means for constructing chart patterns. This is formulated as a chart pattern language (CPL). Figure 3.7 depicts CPL's syntax.

3.4 Definitions of Combinators

In this section, we provide a formal definition of the pattern combinators, together with some of the essential accessor functions to pattern instances. A pattern is interpreted as a function which takes a time interval (b_1, b_2) and returns a (possibly empty) set of *pattern instances*, all of which occur entirely within the time interval in the active price history. Type-wise, we have:

$$
\begin{array}{lll}
\textit{Pattern} & ::= & \textit{Primitive} \mid (\textit{Pattern}) \\
 & & \mid \textit{Pattern}\ \textit{PatOp}\ \textit{Pattern} \\
 & & \mid \textit{Pattern} \bowtie \lambda\, \textit{var}\, .\, \textit{FCExp} \\
\textit{Primitive} & ::= & \textit{bar} \mid \textit{up} \mid \textit{down} \mid \textit{hor} \mid \textit{sprt} \mid \textit{res} \\
\textit{PatOp} & ::= & \gg \mid <> \\
\textit{FCExp} & — & \text{Fuzzy constraint Expressions} \\
\textit{Var} & — & \text{Variables}
\end{array}
$$

Figure 3.7: Syntax of Chart Pattern Language

$\textit{Pattern} :: (\textit{Bar}, \textit{Bar}) \rightarrow \textit{SetOfInstances}$

A naive implementation of $SetOfInstances$ is to make it $[Patt]$, but this is inefficient. We will dwell on this more in Chapter 4.1. In the following, we provide a notation for describing this set of pattern instances.

Definition 1 (Set of Pattern Instances) *Given a pattern* p *and a time interval* (b_1, b_2)*,* $\mathcal{P}(b_1, b_2, p)$ *is the set of all pattern instances of* p *that start from* t_1*,* $t_1 \in [b_1, b_2]$ *and end at* t_2*,* $t_2 \in [b_1, b_2]$*,* $t_2 \geq t_1$*.*

Data type for pattern instances are defined in Figure 3.8. A pattern instance consists of information showing where exactly in the price history does it occur, as well as a fuzzy measure indicating how closely it matches the pattern definition. Note that a bar pattern instance has the constructor named *Bar*; the argument to this constructor is a time value (of type *Bar*). Each of the other primitive pattern instances keeps a pair of bars indicating the starting and ending times of the instance. Constructor *Fb* captures followed-by instances and its component pattern instances which are either primitives or overlay-pattern instances are kept in a list; similarly, constructor *Over* represents overlay instances and keeps a list of its component pattern instances which are either primitives or followed-by pattern instances.

```
data Patt =Bar Bar Fuzzy
          | Up (Bar, Bar) Fuzzy
          | Down (Bar, Bar) Fuzzy
          | Hor (Bar, Bar) Fuzzy
          | Res (Bar, Bar) Fuzzy
          | Sprt (Bar, Bar) Fuzzy
          | Fb [Patt] Fuzzy
          | Ovr [Patt] Fuzzy
```

Figure 3.8: Data Type for Pattern Instances

Associated with a pattern instance are some accessors: Function *fuzzy* takes a pattern instance and returns its fuzzy values. Its definition is trivial and thus omitted. Function *sub* retrieves an instance's sub-components:

```
sub   :: Patt → [Patt]
sub p  = case p of
           (Ovr ps v) → ps
           (Fb ps v)  → ps
           otherwise  → [p]
```

Function *lms* retrieves an instance's landmarks, and is defined as follows:

```
lms    :: Patt → [Bar]
lms p  = case p of
           (Ovr ps v)          → lmsOvr p
           (Fb ps v)           → lmsFb p
           (Bar b v)           → [b]
           (Up (b1, b2) v)     → [b1, b2]
           (Down (b1, b2) v)   → [b1, b2]
           (Sprt (b1, b2) v)   → [b1, b2]
           (Res (b1, b2) v)    → [b1, b2]

lmsFb (Fb ps v) = let (bs : bss) = map lms ps
                  in bs ++ concat (map tail bss)

lmsOvr (Ovr ps v)  =   let lss  = map lms ps
                           ls   = head lss
                           b1   = head ls
                           b2   = last ls
                       in if (length ls) == 1 then ls
                          else b1 : (concat (map cut lss) ++ [b2])
```

Haskell function *tail* returns the tail of a list; *concat* flattens a list of sub-lists; *head* and *last* returns the first and the last element in a list, respectively. Recall that in computing *lms*, we avoid duplication of those landmarks common to multiple component-pattern instances. For followed-by composition, we discard the beginning landmark of all, except the first, followed-by components. For overlay composition, we employ a list function *cut* to discard both the first and the last landmarks of all overlay components. The definition of *cut* is,

```
cut = tail . init
```

tail and *init* are standard prelude functions. Furthermore, we make use of the following fact when computing the landmarks of an overlay pattern instance: Since all components of overlay composition must begin at the same time and end at the same time, if one of the components is a bar primitive (thus consisting of only one landmark), then all component instances must be bar primitives too. In this case, *lms* returns only one landmark.

Constrained Patterns The ($\bowtie$) operation introduces constraints into a pattern. We give a set-theoretic definition of ($\bowtie$) as follows:

$$
\begin{aligned}
&\mathcal{P}(b_1, b_2, p \bowtie f) \;=\\
&\quad \{\, ins1 \mid ins \in \mathcal{P}(b_1, b_2, p) \wedge v = \mathit{fuzzy}\ ins \wedge\\
&\qquad\qquad v' = (\mathit{foldr}\ \mathit{min}\ 1.0\,(f\ \ ins)) \wedge\\
&\qquad\qquad v' > 0.0 \wedge ins1 = \mathit{addFuzzy}\ ins\,(\mathit{min}\ v\ v')\}
\end{aligned}
$$

To perform the constrained-by operation, we first evaluate the argument pattern p to a list of pattern instances. Each of these pattern instance has a fuzzy value denoted by v. We then solve the list of constraints for each of these instances, making use of the knowledge about individual instance. Solving the constraints with respect to an instance produces a list of values, which are then converted to values of type *Fuzzy*. (*foldr min*) operation is then performed on this list of fuzzy values, signifying a conjunction operation. If the result is non-zero, minimum of this result and v is the new fuzzy value for the instance. Function (*addFuzzy*) is defined as follows:

$$
\begin{aligned}
&\mathit{addFuzzy}\ (\mathit{Fb}\ \ \mathit{ps}\ \ \mathit{fz})\,\mathit{fz}' &&= \mathit{Fb}\ \ \mathit{ps}\ \ (\mathit{min}\ \mathit{fz}\ \mathit{fz}')\\
&\mathit{addFuzzy}\ (\mathit{Ovr}\ \ \mathit{ps}\ \ \mathit{fz})\,\mathit{fz}' &&= \mathit{Ovr}\ \ \mathit{ps}\ \ (\mathit{min}\ \mathit{fz}\ \mathit{fz}')\\
&\mathit{addFuzzy}\ (\mathit{Bar}\ \ b\ \ \mathit{fz})\ \mathit{fz}' &&= \mathit{Bar}\ \ b\ \ (\mathit{min}\ \mathit{fz}\ \mathit{fz}')\\
&\mathit{addFuzzy}\ (\mathit{Up}\ \ (b1, b2)\ \ \mathit{fz})\ \mathit{fz}' &&= \mathit{Up}\ \ (b1, b2)\ \ (\mathit{min}\ \mathit{fz}\ \mathit{fz}')\\
&\mathit{addFuzzy}\ \ldots && \quad \text{(Omitted)}
\end{aligned}
$$

From the user's perspective, ($\bowtie$) operation enables him to adapt an existing pattern to the desired one. Operationally, ($\bowtie$) works like a filter. This is expressed by the following properties:[1]

Property 1 *Let p be a pattern and f be the constraint function on pattern instances of p, then*

1. *For any time interval (b_1, b_2),*

   $$\mathcal{P}(b_1, b_2, p \bowtie f) \subseteq \mathcal{P}(b_1, b_2, p).$$

2. *Let g be another constraint function on pattern instances of p, then for any time interval (b_1, b_2),*

   $$\mathcal{P}(b_1, b_2, (p \bowtie f) \bowtie g) = \mathcal{P}(b_1, b_2, p \bowtie (\lambda\, u\, .\, (f\ u) +\!\!+ (g\ u))).$$

Followed-by Composition An instance of the pattern $p_1 \gg p_2$ is obtained by composing an instance of p_1 with another of p_2 such that both instances meet at one point. The fuzzy value of the new instance is the minimum of those of its components. A set-theoretic definition of the $(\gg)$ operation is as follows:

$$\begin{aligned}
&\mathcal{P}(b_1, b_2, p_1 \gg p_2) = \\
&\quad \{(\mathit{fby}\ \mathit{in}1\ \mathit{in}2\ v) \mid \\
&\qquad \mathit{in}1 \in \mathcal{P}(b_1, b_2, p_1) \wedge \mathit{in}2 \in \mathcal{P}(b_1, b_2, p_2) \wedge \\
&\qquad (\mathit{last}\ (\mathit{lms}\ \mathit{in}1)) = (\mathit{head}\ (\mathit{lms}\ \mathit{in}2)) \wedge \\
&\qquad (v_1 = \mathit{fuzzy}\ \mathit{in}1) \wedge (v_2 = \mathit{fuzzy}\ \mathit{in}2) \wedge \\
&\qquad v = \mathit{min}\ v_1\ v_2\}.
\end{aligned}$$

Function *fby* combines the two component instances of followed-by operation to yield a followed-by instance. It is defined as follows:

[1] The Haskell function $+\!\!+$ concatenates two lists.

```
$fby :: Patt \rightarrow Patt \rightarrow Fuzzy \rightarrow Patt$
$fby\ p1\ p2\ v =$
  case $(p1, p2)$ of
   $(Fb\ cs1\ v1,\ Fb\ cs2\ v2)$ $\rightarrow Fb\ (cs1 +\!\!+ cs2)\ v$
   $(Fb\ cs1\ v1,\ p2)$          $\rightarrow Fb\ (cs1 +\!\!+ [p2])\ v$
   $(p1,\ Fb\ cs2\ v2)$          $\rightarrow Fb\ (p1 : cs2)\ v$
   $(p1, p2)$                    $\rightarrow Fb\ [p1, p2]\ v$
```

Since a followed-by instance maintains a list of components that have been composed by followed-by operations, the $(\gg)$ operation is therefore associative. Formally,

Property 2 *Let p_1, p_2 and p_3 be three patterns. For any time interval (b_1, b_2),*

$$\mathcal{P}(b_1, b_2, (p_1 \gg p_2) \gg p_3) = \mathcal{P}(b_1, b_2, p_1 \gg (p_2 \gg p_3)).$$

Overlay Composition An instance of the pattern $p_1 <> p_2$ is obtained by composing an instance of p_1 with another of p_2 such that both instances meet at *both* the start and the end point. Again, the fuzzy value of the new instance is the minimum of those of its components. A set-theoretic definition of the $(<>)$ operation is as follows:

$$\begin{aligned}
&\mathcal{P}(b_1, b_2, p_1 <> p_2) = \\
&\quad \{(overlay\ in1\ in2\ v)\ | \\
&\qquad in1 \in \mathcal{P}(b_1, b_2, p_1) \wedge in2 \in \mathcal{P}(b_1, b_2, p_2) \wedge \\
&\qquad (head\ (lms\ in1)) = (head\ (lms\ in2)) \wedge \\
&\qquad (last\ (lms\ in1)) = (last\ (lms\ in2)) \wedge \\
&\qquad (v_1 = fuzzy\ in1) \wedge (v_2 = fuzzy\ in2) \wedge \\
&\qquad v = min\ v_1\ v_2\}.
\end{aligned}$$

Function *overlay* combines the two component instances of an overlay operation to yield an overlay instance. It is defined as follows:

```
  overlay :: Patt → Patt → Fuzzy → Patt
  overlay p1 p2 v =
    case (p1, p2) of
     (Ovr cs1 v1, Ovr cs2 v2) → Ovr (cs1 ++ cs2) v
     (Ovr cs1 v1, p2)         → Ovr (cs1 ++ [p2]) v
     (p1, Ovr cs2 v2)         → Ovr (p1 : cs2) v
     (p1, p2)                 → Ovr [p1, p2] v
```

Just like the $(\gg)$ operation, $(<>)$ operation possesses the associative property:

Property 3 *Let p_1, p_2 and p_3 be three patterns. For any time interval (b_1, b_2),*

$$\mathcal{P}(b_1, b_2, (p_1 <> p_2) <> p_3) = \mathcal{P}(b_1, b_2, p_1 <> (p_2 <> p_3)).$$

3.5 Pattern Reusability

For CPL to be acceptable to the financial analysts, it is necessary to provide a library of commonly-used patterns. Therefore, reusability of patterns becomes crucial as users may want to refine and compose these library patterns. It is unreasonable for users to be unduly concerned about the exact sequence of sub-components in the pattern definition. Users should be able to access the landmarks and sub-components of a pattern in a standard order, regardless of the exact sequence of composition of the pattern.

For example, consider the use of a diamond pattern already defined in the system library. While the user is aware of the fact that a diamond is composed of overlaying two resistance lines on two support lines, he should not be concerned of whether the diamond is constructed by

$$(res \gg res) <> (sprt \gg spt)$$

or

$$(sprt \gg sprt) <> (res \gg res)$$

However, these two compositions will produce different ordering of landmarks and components, if the user were to access them via *lms* and *sub* functions. To overcome this problem, we propose a standard ordering for component-pattern instances and landmarks.

Note that this problem does not occur in case of followed-by operation, since $\gg$ is not commutative. The ordering the landmarks and sub-components of a followed-by instance is always from left to right or in other words in chronological order of their occurrence. Therefore, we focus on ordering the components and landmarks of overlay patterns only.

Next, we observe that once the component-pattern instances are properly ordered, landmark ordering problem will no longer exist. In other words, having a standard ordering on the components is sufficient to ensure a standard ordering on the landmarks. Consequently, we restrict our attention to the ordering of component-pattern instances only.

So, exactly what constitutes a good ordering of component-pattern instances? We believe that a good ordering is one that can be determined uniquely from the pattern definition, *not* the pattern *instance*. By fixing an ordering on a pattern, we treat all its instances in a uniform way.

Thus, we define an ordering over pattern instances as follows:

Definition 2 (Pattern Instance Ordering)

1. *Pattern instances are ordered based on their data constructors, as follows:* $Bar < Res < Up < Hor < Down < Sprt < Fb < Ovr$.

2. *Two composite instances with the same data constructor are ordered by lexicographic ordering of their lists of components.*

3. *Otherwise, two pattern instances are of equal order.*

For example,

$$(Res\ (2,10)\ 1\ 0.4) < (Up\ (4,5)\ 0.6) < (Fb\ [..]\ 0.3)$$
$$Fb\ [Res\ (2,5)\ 1.5\ 0.4, ..] < Fb\ [Sprt\ (1,5)\ 0.5\ 0.3, ..]$$

In CPL, we have a built-in function *orderPat* which takes a pattern instance and returns a new instance such that sub-components of its overlay components are recursively ordered according to this rule. *orderPat* is defined as follows:

```
orderPat :: Patt → Patt
orderPat p =
 case p of
 (Fb ps v)   → Fb (map orderPat ps) v
 (Ovr ps v)  → orderOvr p
 otherwise   → p
 where orderOvr (Ovr ps v) =
           let ps1 = map orderPat ps
               ps2 = sortBy ovrOrder ps1
           in Ovr ps2 v
```

Haskell function *sortBy* takes a comparison function and a list of data, and sorts the list using the comparison function. *ovrOrder* is the comparison function defined according to Definition 2. The detail is omitted.

As an example, consider applying *orderPat* to the following pattern instances (we show each component by its data constructors only):

$$Ovr\ [Fb\ [Res,\ Ovr\ [Up, Sprt]],\ Fb\ [Res,\ Ovr\ [Down,\ Res]]]$$

The resulting pattern instance is of following structure:

$$Ovr\ [Fb\ [Res,\ Ovr\ [Res,\ Down]],\ Fb\ [Res,\ Ovr\ [Up, Sprt]]]$$

Notice that in the resulting pattern instance, the components of all the overlay *sub*-patterns follow the pattern instance ordering.

Now, for our diamond example, no matter how *dmnd* is defined, the user can reuse the pattern as follows:

$$\begin{array}{l} dmnd\ \bowtie\ \lambda\, u\ .\ \mathbf{let}\ \begin{array}{lcl} u' & = & orderPat\ u \\ {[rs, ss]} & = & sub\ u' \end{array} \\ \qquad\qquad\quad \mathbf{in}\ \ldots \end{array}$$

There is another advantage for ordering the components based on pattern definition rather than pattern instances: It enables the ordering information to be largely computed at compile time. On the other hand, there are limitations to overlay ordering:

1. It does not uniquely order two component-pattern instances which are constructed by the same primitive, but with different constraint functions.

2. It may not uniquely order a pattern instance with dynamic component against another pattern. For example, the following two patterns cannot be ordered: (*if* ... *then res else sprt*) and *up*.

For those patterns with ambiguous ordering, user has to know how they are constructed to use *lms* and *sub* operations.

3.6 Related Work

CPL is a programming language for specifying geometrical patterns in time-series databases. In CPL, patterns are specified using syntactic constraints (in terms of primitive patterns) and semantic constraints (in terms of a set of constraints). On its own, syntactic pattern specification and its recognition techniques have been studied extensively in [13]. Here, patterns are described using some primitive patterns, and the recognition techniques translates the entire pattern into a string of tokens (primitive curves) for matching.

The syntactic constraint of CPL is related to the *Shape Description Language* (SDL) proposed by Agrawal *et al.* [2]. In SDL, patterns are described in term of primitives like up, down, stable, *etc.*, each of which are one time-unit long. Patterns can then be composed from these primitives in the form of regular expressions. SDL does not handle overlay patterns. Furthermore, it suffers from the following limitations: Firstly, SDL takes a microscopic approach in defining patterns. As such, it is difficult to express in SDL the global features of a pattern using these primitives, which are local in nature. For example, a triangle pattern is formed by a pair of support and resistance line, ignoring the small fluctuations within those line. Defining triangle using above microscopic primitives can become a complex task. Secondly, CPL has a strong support for creating indicators and using them in pattern definitions. An Indicator is actually a new time series (eg. moving average) created out of raw a time series like closing prices. So it is very useful to be able to define patterns not only on the raw time series, but a new transformed series created out of it. SDL does not have such a facility.

The semantic constraints of CPL are inspired by the landmark model proposed by Perng *et al.* for similarity-based pattern matching [26]. Briefly, this model is built from study of human visual system, which finds that much of the visual data perceived by human being is redundant; some dominant points on the shape contour are rich in information content, and are sufficient to characterize a shape [7, 19]. Consequently, we define landmarks in a pattern to be those points having important shape attributes; *i.e.*, those that form the joints of the skeleton of a pattern. By providing constraint on the landmarks, patterns can be defined from a macroscopic level, ignoring unnecessary details.

Our choice of pattern-composition operations allows us to support interval-based temporal reasoning of price history. These operations are closely related to those defined in Allen's algebra for modeling times and events [3]. Allen's algebra (as well as its associated operators) has been used widely in developing query languages for temporal databases, such as TQUEL [31], TSQL [24], *etc.* However, these languages mainly translate Allen's operations into point-based temporal query; they do not make full use of interval-related temporal properties in their implementation. On the other hand, CPL as presented so far only supports two of the Allen's operators; we are currently extending CPL to fully support Allen's operators.

Lastly, we have modeled technical indicators as functions, similar to *Behaviors* in Fran [11] and *Observables* in the work on Financial contract [27]. Since technical indicators operate on history data of a security, which can be viewed as a list of data, it is possible to model indicators as operations over infinite streams, as in LUSTRE [17]. However, the treatment of patterns as illustrated in this paper shows that we usually work on a finite period of time, and many operations — such as accessing a pattern's landmark — do not access history data in chronological order.

Chapter 4

Implementation

4.1 Causes of Inefficiency

Embedding CPL within Haskell provides an expressive algebraic structure with intuitive syntax to the user, for free. Whereas, getting an efficient implementation demands more expensive endeavor.

The primary objective of the interpreter is to retrieve all pattern instances occurring in price histories that match a pattern definition. As we have already seen, patterns are defined using constraints on their landmarks and constituting sub-patterns. So searching for pattern instances in price history involves satisfying those constraints; in other words, a good interpreter calls for an efficient constraint solver.

A naive implementation of the interpreter uses Haskell list to represent the set of pattern instances, which matches the semantics of CPL(as described in Section 3.4) very nicely. But its performance leaves much to be expected (the program does not terminate for hours) when we feed it with 10 years data of 30 companies (Each yearly data has daily information of 5 basic technical indicators.)

By profiling the performance of the naive interpreter, we have identified three major bottlenecks, which must be addressed to achieve the desired efficiency. These three bottlenecks are: Inefficient constraint solving; frequent analysis on layers of data types which have been used to enforce strong typing; and inefficient reading of huge amount of price histories from the file.

Inefficient constraint solving can be attributed to modeling of domain specific entities, like patterns as functions which inhibit their inspection, which is required for optimization. It also precludes any scope for their optimizing transformations. In this chapter, we present an interesting solution to this problem of inspecting functions by exploiting the advance type system features like *type classes* and *local quantification annotation.* Although the intention of our function inspection is typical to our problem domain, ie it facilitates efficient constraint solving, we believe that the approach is general and can be applied to other domains.

4.2 Choosing A Tree Structure

As we mentioned in the previous chapter, patterns are modeled as functions of type `Pattern`, which is defined as follows,

```
type Pattern = (Bar, Bar) -> SetOfInstances
```

We have deliberately chosen to use a different font from previous chapter to show the Haskell codes, to emphasize that codes appearing in this chapter are actually executable, whereas in the previous chapter they represented specification. Notably, we used $\gg$, $<>$, $\bowtie$ for followed-by, overlay and constraint-by operations, but in actual implementation they are replaced by `>.>`, `|.|` and `?` respectively.

The first efficiency consideration is to choose an efficient data structure for `SetOfInstances`. A naive representation of it, is a list of pattern instances, `[Patt]`(`Patt` was defined in Figure 3.8) This representation leads to redundant computation during searching of pattern instances. For example, consider the following pattern definition,

```
(up >.> down) ? \p-> let [a,b,c] = lms p
                     in [(low b < low a)]
```

To evaluate this pattern for an interval, say `(2,10)`, we first find all instances of `(up >.> down)` within the interval `(2,10)`. Suppose `up`, `down` patterns give following pattern instances when evaluated for the interval `(2,10)`,

```
[Up (2,4), Up (2,6), Up (2,7), Up (3,6), Up (3,7)]
[Down (6,8), Down (6,9), Down (7,9), Down (8,9)]
```

Evaluating `>.>` with above instances will generate a list of instances-

```
[Fb [Up (2,6), Down (6,8)], Fb [Up (2,6), Down (6,9)],
 Fb [Up (3,6), Down (6,8)], Fb [Up (3,6), Down (6,9)],
 Fb [Up (3,7), Down (7,9)]]
```

Next, for each of these instances we check if the constraint `(low b < low a)` holds. Suppose that `(Up (2,6))` with `a=2, b=6` fails to satisfy the constraint. In that case, both the first and the second instances in the above list will not be in the final result since they both contain `(Up (2,6))`. However, since constraint satisfaction is performed on each instance, the constraint on `(Up (2,6))` will be checked twice instead of one.

To avoid this unnecessary checking of constraints, we employ a tree structure for the set of pattern instances, such that instances whose list of landmarks have common prefix are grouped under one subtree, as shown in Figure 4.1 for the above instances. Now, when the constraint solver fails to satisfy a constraint using some prefix, such as `Up (2,6)`, it will avoid checking all instances that have the same prefix. Therefore, we use list of trees as an efficient implementation of `SetOfInstances`.

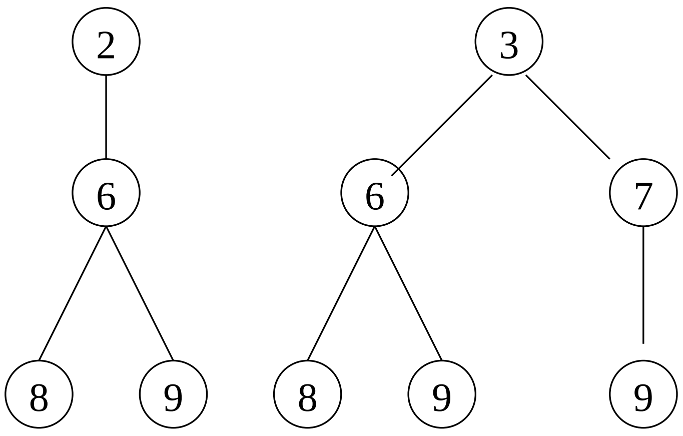

Figure 4.1: Trees of instances

```
type SetOfInstances = [Tree]

data Tree = Tree Bar [Tree]
```

Each tree in the list represents a set of instances that have the same starting landmark, ie. the value store in the root of the tree. We refer these trees as *pattern-instance* trees.

For efficiency, we enforce two invariants on this list of trees:

- No two trees in the list have roots having same value.
- The list is sorted in increasing order of values of their roots.

These two invariants make the implementation of `(>.>)` operator efficient in the following way. Here is an informal algorithm for p_1 `>.>` p_2 operation:

```
For each instance x in SetOfInstances of p_1
  find all instances y in SetOfInstances of p_2
      such that y begins from the end point of x
```

In case, where the trees were not sorted, in order to find an instance of p_2 that can be joined to the end of an instance of p_1, our interpreter would, in the worst case, have to scan *all* the trees in the list of p_2. However, when the trees are sorted according to the values of their root, we need only to scan the list until we encounter a tree with root of value greater than the end point of the p_1 instance under consideration. The first invariant strengthens the above terminating condition for scanning the second list. It assures that it is correct to stop scanning further when we find a tree in the second list that can be joined with a p_1 instance (we would not find another tree with same root from further scanning.)

4.2.1 Associativity: Left Vs. Right

In a `(>.>)` operation, the two operand sub-pattern instances are combined by matching the ending landmark of the left-operand against the starting landmark of the right-operand (they should be same.)

End landmarks of a set of instances that are stored in the same pattern-instance tree are the leaves of the tree. And the root of a tree is the first landmark of all the instances represented by that tree. So for `(>.>)` operation, for each leaf of the left-operand tree, we find a tree from the list of right-operand trees whose root has the same value as that of the leaf. It means that, the number of times the list of right-operand trees is scanned, is proportional to the number of leaves of the left-operand tree. Under such condition, if we were to make `(>.>)` left associative, as the interpreter joined a series of trees from left to right, the number of leaves in the combined left tree would multiply rapidly, resulting in more scan needed on the right-operand trees. Hence, by *defining* `(>.>)` *to be* **right associative**, we effectively cut down the number of scans over the right-operand trees.

4.3 Constraint Solving

The general form of pattern specification is p ? f. For this form, our pattern-searching algorithm does the following: (1) It searches the sub-patterns of p in a *divide-and-conquer* fashion, yielding a set of pattern instances for p; (2) it performs backtracking on the set of pattern instances by solving the constraints encapsulated in f.

4.3.1 Lazy Divide-and-Conquer Constraint Solving

As we have seen in previous examples, patterns are often composed of sub-patterns using `(>.>)` and `(|.|)` operations. So intuitively, to search for pattern instances, we first search for the instances of its sub-patterns, which are later combined to form the instances of the original pattern. Effectively, we have adopted a divide-and-conquer (D&C) approach to search for pattern instances, where the division of the problem is determined from the users' pattern definition.

Lazy evaluation of Haskell adds a new flavor to the divide-and-conquer approach; it conquers the subproblems lazily, if and when required. The requirement arises when we combine the solutions of the sub-problems. In our application, suppose we have a pattern `(a >.> b)`. When we evaluate the subproblems ie. find the instances of sub-patterns `a` and `b`, we do not find all instances of them first, rather their evaluation is driven by `>.>` function that consumes the instances of `a` and `b`. As a result, we may not evaluate some of the instances of the sub-patterns. As an example, if none of the instances of sub-pattern `a` end at bar(day) `n`, lazy evaluation takes care not to find any instances of `b` that start at `n`.

In our application, divide-and-conquer was the natural choice of evaluation, given that patterns are written in a compositional style. And as have we pointed out, lazy evaluation makes this approach more effective. Although natural in our case, finding pattern instances is operationally constraint solving. This suggests that our approach is applicable to normal constraint solving. We have applied this algorithm to N-queen problem and reported our results in [4]. Next chapter presents this algorithm in more detail.

4.3.2 Constrain-by Operator (`?`)

For a pattern definition of the form p `?` f, first we find all the instances of p and then for each of them we check the constraints f. The patterns instances, as we described in Section 4.2, are represented as trees whose nodes contain the landmarks of the instances, and instances with same prefix landmarks share a common branch. So checking the constraints for these tree structured pattern instances involves traversing the tree depth-first and at each level(node) checking the constraints that involve landmarks constituting the path from root upto that node. And if in any node, the constraint is not satisfied, there is no benefit going down further, instead we backtrack and take another branch. So effectively, we do a simple backtracking on the pattern-instance trees. The result of this backtracking is a pattern-instance tree similar to the input tree, sans those branches that do not satisfy the constraints.

As simple it might sound, implementation of this naive backtracking on pattern-instance tree turn out to be involved. Because CPL is a “truly” embedded language in Haskell, we can not analyze the pattern definition. So we shall start with an intuitive definition of backtracking and later point out two problems and present their solutions with the final and correct version of backtracking.

```
backtrack cons = (prune id) . (check cons)
```

Function `backtrack` takes a constraint function `cons` (a list of constraints) and a pattern-instance tree and returns a tree with those branches that do not satisfy the constraints pruned off. As its name suggests, `check` does the constraint check and annotates each node in the tree with a boolean flag denoting if the path leading to it(corresponding pattern instance) satisfy the constraints. `prune` takes in such an annotated tree and prunes off all subtrees in the tree whose root has been flagged `False`. We would like to point out that due to lazy evaluation model these pruned subtrees are never evaluated beyond its root.

Now we look into the implementation of `check` function. To annotate each node in the tree with a boolean flag, it has to find out which of the list of constraints can be evaluated at that level. Those constraints that can be evaluated that will involve *only* landmarks found on the path from the the root to the current node. It should not try to evaluate constraints that require the values of the landmarks that correspond to levels below the current level. For example, consider the following definition,

```
patt = ... ? cons
  where cons =
        \p -> let [a,b,c,d] = lms p
              in [close b > close c,      -- (1)
                  high a == high d,       -- (2)
                  open a - open c < 10 ]  -- (3)
```

In the above definition, the constraint `1` and `3` can be checked when we have reached the third level (The root is considered to be at level 1 and contain the value of landmark `a`) of the tree as the values of landmarks `a`, `b` and `c` become known. Whereas constraint `2` can only be checked at level `4` when we have the value of landmark `d`. But given that, individual constraints are of type `Bool`, `check` has no ways to find out this syntactic information during runtime, which is our first problem. This leads us to the realization that for `check` to work, it needs some more information about the constraints. So we change our `backtrack` definition as follows,

```
backtrack cons = (prune id) . (check cons indx)
```

```
        where indx = ...
```

Where `indx` is a list of type `[(Int,[Int])]`. An element in this list of the form $(i, [j_1, j_2, \ldots, j_n])$ represents that information available at a node at i^{th} level of a tree (values of 1^{st} to i^{th} landmarks) is sufficient to evaluate the j_1, j_2, $\ldots$, j_n^{th} constraints in the `cons` function. We defer the description of calculation of `indx` to next section.

Before we introduce the second problem, we explain how `check` evaluates the constraints encapsulated in `cons`. `cons` has the following type,

```
cons :: Patt -> [Bool]
```

To evaluate the constraints, `check` calls `cons` function with a pattern instance as argument and receives from `cons` a list of boolean values, which denotes the satisfaction of each constraint for the input pattern instance; for a resultant list of booleans of the form $[b_1, b_2, \ldots, b_k]$, b_i represents if i^{th} constraint is satisfied. But the pattern-instance tree contains the landmarks, so where does `check` get a pattern instance to pass to `cons`? This is done by a supporting function called `toPatt` which conjures up a pattern instance (of type `Patt`) from a list of landmarks (of type `[Bar]`). But unfortunately, `toPatt` requires a complete list of landmarks to come up with the pattern instance. By complete list we mean, if the pattern has, say, `4` landmarks, `toPatt` requires that its input list has `4` elements. This brings us to the second problem: in the above example, suppose we are at the level `3` of the tree, and hence have enough information (values of landmarks `a`, `b` and `c`) to evaluate the third constraint. But we can not pass a list of values of `a`, `b` and `c` to `toPatt`; it needs a list of `4` values. In the following, we present two solutions to this problem, both using laziness.

In the first solution, `toPatt` function substitutes `undefined` in place of all unknown landmarks. `undefined` stands for bottom in Haskell. Since invoking `cons` at any level of the tree, will not attempt to evaluate constraints involving unknown landmarks (that corresponds to levels below the current level), `undefined` landmarks will never be touched. Although this solution is simple, it is inefficient. It involves a call to `toPatt` function at every node of the tree. We alleviate this inefficiency in our second solution as follows.

We introduce a function `tTransform` that takes in the pattern-instance tree, where the nodes contain values of landmarks, and returns a tree where each node is replaced by the path from the root to the leftmost leaf of the subtree rooted at this node. Figure 4.2 gives a pictorial example of the operation of `tTransform`.

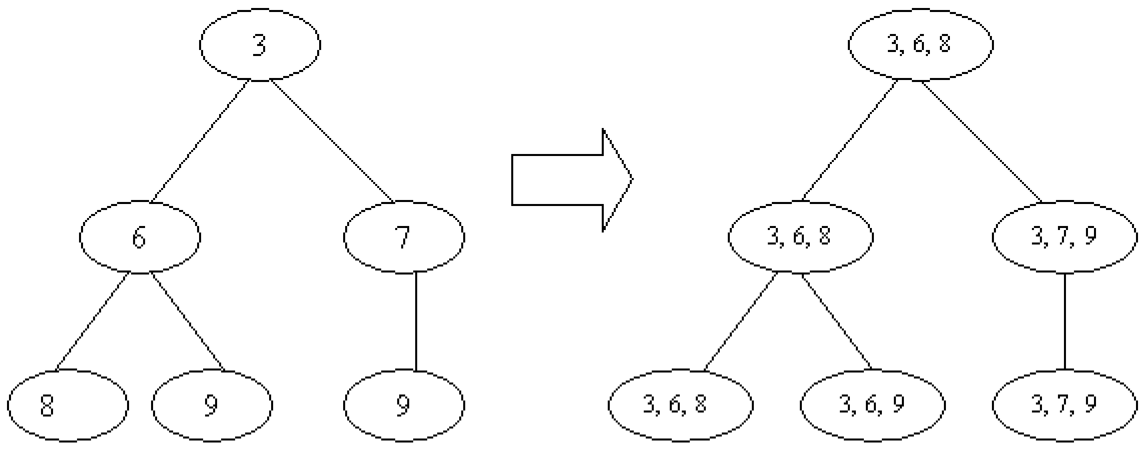

Figure 4.2: `tTransform`

How does `tTransform` help? In our first solution, we used to replace `undefined` in place of all unknown landmarks. Here we will instead use the list of values of landmarks contained in a node as the argument to `cons`. And the advantage is: as is obvious in Figure 4.2, in the resulting tree the left child of a node has the same value as itself. This implies that when all the relevant constraints are satisfied for a node, we do not need to call `cons` again for its left child, instead we can use the list of booleans values returned by `cons`; and we check the particular elements of this list corresponding to constraints that can be checked in its left child.

With regard to the conformance of this solution to backtracking algorithm, lazy evaluation takes care of that. Although each node contains a complete list of landmarks starting from root to the leftmost leaf of the node, Haskell maintains thunks for the values of landmarks from a node's left child to its leftmost leaf; they are explored only on demand.

Incorporating `tTransform` into our `backtrack` function we get,

```
backtrack cons = (prune id) . (check cons indx) . tTransform
    where indx = ...
```

Figure 4.3 shows the complete listing of the functions described above.

4.3.3 Indexing

We now turn our attention to `indx` calculation. `cons` takes a pattern instance (complete path) and returns the result of all the constraints. Had these constraints been represented as first-order data types, we would have done some manipulation to find which constraint involve which variables. Functional representation of constraints is natural and simple in one hand, but poses this problem of disallowing any analysis on it on the other hand. Here we present a solution to this problem, without changing the functional representation of constraints. Let's consider the constraint function of the previous example:

```
cons = \p -> let [a,b,c,d] = lms p
             in [close b > close c,      -- <3>
                 high a == high d,       -- <4>
                 open a - open c < 10 ]  -- <3>
```

The first constraint involves the second and third landmarks. This means that the constraint can be evaluated when these two landmarks have been instantiated. In other words, at the 3^{rd} level of a tree we will have enough information (partial solution) to verify (the satisfiability of) this constraint. Using similar reasoning, we find that the second and third constraints can be verified at levels 4 and 3 of a tree, respectively. These level information are listed to the right of the constraints.

Informally, in order to find out these levels, we first replace the landmark variables (*i.e.*, `a,b,c,d`) by their positions in the list of variables (*i.e.*, [a,b,c,d]). Thus, `a`, `b`, `c`, and `d` should be replaced by 1, 2, 3 and 4 respectively. Next, for each constraint, we find the maximum of those replaced values used in the constraint. For example, in the second constraint above, we replace `a` and `d` by 1 and 4 respectively, and the maximum of them is thus 4. We refer this resulting number as the *"deepest involved variable"* of a constraint, since for example, the variables (landmarks) involved in the second constraint are `a` and `d` and `d` is situated *deeper* than `a` in the tree, which makes `d` (infact its index in the list of landmarks ie. 4) as the *deepest involved variable.*

Once we compute list of *deepest involved variable*s corresponding to each constraint (`[3,4,3]` in this case), we can compute `indx` by applying `index'` function to this list. It returns `[(1,[]), (2,[]), (3,[1,3]), (4,[2])]` in our example, which implies- there are no constraints that can evaluated at level 2 and at level 3, we can check the constraints `1` and `3`.

```
index' :: [Int] -> [(Int,[Int])]
index' divs = foldl (\acc d ->
                        if null (d `lookup` acc)
                        then (d, elemIndices d divs):acc
                        else acc) [] divs


lookup' :: (Ord a) => a -> [(a,[b])] -> [b]
lookup' x ((y,bs):rest) | x == y    = bs
                        | otherwise = lookup' x rest
```

```
lookup' _   []     = []
```

This analysis is very intuitive, and Implementing this in a compiled language is straight forward, but in an interpreted language it needs some tricky manipulation as follows.

Consider one of the three constraints of the previous example,

```
    open a - open c < 10
```

The *deepest involved variable* for this constraint is 3 (corresponding to landmark `c`), which we aim to find out in an interpretive mode. Recall that `open` is one of the basic technical indicators, with the following type,

```
type Indicator a = Int -> Maybe a
open :: Indicator Float
```

For our analysis, we change the `indicator type` to as follows,

```
type Indicator a = Value Bar -> Value a
open :: Indicator Float

data Value a = Value a | DIV Int | Nope
```

Following table gives the correspondence between the behavior of the new declaration and the original one. To get the opening price of day 3 for example,

```
                       original      new
we evaluate:            open 3      open (Value 3)
it may return:          Just 5.6    Value 5.6
or it may reutrn:       Nothing     Nope
```

Apart from the above, the new `open` function also takes `(DIV n)` and returns `(DIV n)`; behaves like an identity function. We also instantiate the `Num` class of Haskell for the `(Value a)` type, which supports all arithmetic operations for it.

```
instance (Num a) => Num (Value a) where
  (Value i) + (Value j) = Value (i + j)
  (Value i) + (DIV j)   = Value (i + j)
  (DIV i)   + (Value j) = Value (i + j)
  (DIV i)   + (DIV j)   = DIV (i `max` j)  -- Tricky line
    _       +   _       = Nope
  ...
  fromInteger x = DIV 0           -- One more trick

(>), (<), ... :: (Ord a) => Value a -> Value a -> Value Bool
-- We hide the prelude definition of these
-- by doing  a qualified import of prelude
(Value i) > (Value j) = Value (i > j)
(Value i) > (DIV j)   = Value (i > j)
(DIV i)   > (Value j) = Value (i > j)
(DIV i)   > (DIV j)   = DIV (i `max` j)
  ...
```

Now we return to our example of finding the *deepest involved variable* for the constraint `open a - open c < 10` and see how the above changes will lead us to a solution. Suppose in this constraint, we have `a = (DIV 1)` and `c = (DIV 3)`, then according to the above declarations, evaluation of this constraint will yield `(DIV 3)`, which we interpret as, *deepest involved variable* for this constraint is `3`.

But how do the landmarks `a` and `c` get the values `(DIV 1)` and `(DIV 3)`? Looking at the definition of `cons` in the example, we see that the values of `a` and `c` are returned by the `lms` function. `lms` has the type `Patt -> [Bar]`. We introduce few more constructors (called *tricky constructors*) into `Patt` type as follows. `lms` definition is also changed to handle these new constructors.

We change `Patt` type which we change as following,

```
Patt = Up (Bar,Bar) Fuzzy
      |...
      | UpT | DownT | FbT [Patt] | ... -- Tricky Constructors


lms :: Patt -> [Value Bar]
lms Up (b1, b2) = [b1,b2]
...
lms  p   = map DIV [1..(count p)]
    -- p is one of the tricky constructors
    -- like UpT, DownT, FbT ..
   where count UpT       = 2
         count DownT    = 2
         count (FbT ps) = sum (map count ps) + length ps - 1
```

Now when `lms` is input with `(FbT [UpT, DownT])`, it returns `[DIV 1, DIV 2, DIV 3]`. Also, it returns `[Value 5, Value 7, Value 9]` for input `(Fb [Up (5,7) 0.2, Down (7,9) 0.1])`.

So to summarize the whole picture, there are two phases in constraint solving: (1) calculating `indx` (2) actual constraint solving. To calculate `indx`, constraint-by operator `(?)` passes a *dummy pattern instance* to the constraint function instead of the actual instance, which triggers the evaluation at `DIV` *level* to compute `indx`. A *dummy pattern instance* is made up of *tricky constructors* and is structurally similar to the actual instance. For example, `(FbT [UpT, DownT])` is a *dummy pattern instance* for `(Fb [Up (5,7) 0.2, Down (7,9) 0.1])`. In second step, `backtrack` uses `indx` and checks for the constraints on the pattern-instance tree. During this phase, `backtrack` invokes the constraint function with actual pattern instance as argument.

The problem we addressed here can be categorized as, *on-the-fly function inspection and analysis.* We believe that the solution is interesting, and may be applicable to other applications, especially for embedded languages, where this need to inspect functions arises often. But unfortunately, the solution is computationally inefficient, as is evidenced from the following profiling:[1]

```
total time  = 49.74 secs   (2487 ticks @ 20 ms)
total alloc = 5,482,674,952 bytes
              (excludes profiling overheads)


COST CENTRE          MODULE     %time %alloc


parse                Mutables    46.5   41.8
GC                   GC          24.1    0.0
?!                   Mutables     8.6    5.7
lift2V               Operators    6.0    4.6
```

From the above profile, we realize there are two culprits. Firstly, the `parse` function which parses the data from text format to floating point values takes up a lot of time and space. Function (?!) is the indexing operator used to access data from the arrays. Since constraint solving involves a lot of such access, it is not surprising that it takes a significant time. The second culprit is the `lift2V` function that is defined below. It supports the binary operations on the `(Value a)` type that encompass `DIV` and `Value` data constructors.

```
lift2V :: (a->b->c) -> (Value a -> Value b -> Value c)
```

[1]All the performance statistics presented in this paper are run under Linux in a desktop PC with P4 1.7GHz processor and 256 Mbytes of RAM. The interpreter is called to search for all head-and-shoulder pattern instances occurring in 10 years of daily price histories of 40 companies. The head-and-shoulder pattern is defined in Appendix A.

```
lift2V op (Value x) (Value y) = Value (x ‘op‘ y)
lift2V op (DIV x) (DIV y) = DIV (x ‘max‘ y)
....
```

Any operators that works on the `(Value a)` data type will have to perform pattern matching over these data constructors. Although we use `DIV` data constructor only during index computation, we have to pay for its overhead throughout constraint solving. This degrades the performance rapidly, since constraints generally consists of a lot of binary operations on values of this type.

To overcome this computational inefficiency, instead of putting the *tricky data constructors* in the `Patt` type, we create a new data type for them, called `PattT`. Similarly, we introduce a new type `DIV`, to have the `DIV` constructor separated from `Value`.

```
data PattT = UpT | DownT | FbT [PattT] | ...
data DIV = DIV Int
```

Next, we introduce the following classes, to handle these new types.

```
class (Fractional b) => Indicator a b | a -> b where
  close, open, low, high :: a -> b
instance Indicator Bar (Maybe a) where
  ...
instance Indicator DIV DIV where
  ...

class LMS a b | a -> b where
  lms :: a -> [b]
instance LMS Patt Int where
  ...
```

```
instance LMS PattT DIV where
  ...

class (Logic b) => Compare a b | a -> b where
 (>),(<),(==),(<=),(>=),(<>) :: a -> a -> b
instance Compare Price Bool where
  ...
instance Compare DIV DIV where
  ...
```

With this setting, we see that constraint function is overloaded to handle both `PattT` and `Patt` types. `backtrack` function will use both versions of constraint functions; *i.e.*, one that takes `PattT` when evaluating `indx`, and another that takes `Patt` inside `check` function. The normal Haskell Type system does not allow co-existence of these two versions of constraint function. But we overcome this problem by introducing **local quantification type annotation** as follows. This helps the type system to decide which version of the function to choose at each of the function's call sites.

```
backtrack ::
 (forall a b c d. (LMS a b, Indicator b c, Compare c d) => a -> [d])
 -> Tree -> Tree


   -- constraint_function :: PattT -> DIV    , OK
   -- constraint_function :: Patt  -> [Bool] , OK
```

The following profile shows the improvement after separating the *tricky constructors* from the actual constructors into two different types, eliminating the overhead incurred by `liftV` function.

```
total time  = 36.16 secs   (1808 ticks @ 20 ms)
total alloc = 3,941,450,740 bytes
              (excludes profiling overheads)

COST CENTRE          MODULE     %time %alloc


mypars               Mutables    61.3   58.1
GC                   GC          27.9    0.0
?!                   Mutables     8.9    7.9
?                    Pattern      8.4   10.7
```

In summary, we have accomplished our initial goal of constructing a backtracking algorithm to traverse the pattern-instance tree and check the constraints efficiently. As we saw, this led to us few problems due to the interpretive setting. Functions do not allow inspection. To this, we presented an interesting solution. And we addressed the inefficiency of our initial solution by using features of Haskell type system. We took full advantage of laziness of Haskell in our solutions.

4.4 Efficient Inputing

As mentioned earlier, the input price history consists of 5 values. Since Haskell's standard mechanism for I/O allows these data to be read only in a textual representation, the standard way to store price histories in files is to keep them in ASCII representation. Considering accessing ten-year price histories of 300 companies, one can imagine the amount of time spent on translating these data to floating points.

Inspired by the result reported by Wallace and Runciman on heap compression and use of binary I/O [32], we store these price histories in binary, and make use of Haskell's binary I/O to read in the data. The following profile shows the dramatic improvement due binary IO. Notice that the function `mypars` (responsible for parsing), visible in previous profile, does not appear among the top time-consumers here.

```
total time  = 13.52 secs   (676 ticks @ 20 ms)
total alloc = 1,962,840,344 bytes
              (excludes profiling overheads)

COST CENTRE          MODULE     %time %alloc


GC                   GC          26.2    0.0
cons                 CPL         21.0   23.6
?!                   Mutables    19.5   12.8
```

4.5 Related Work

In this chapter, we presented the efficiency bottlenecks and their solutions in CPL implementation. There are two main contributions: A novel constraint solving algorithm and on-the-fly analysis and optimization of pattern definitions. We will present the works related to constraint solving algorithm in the next chapter.

Embedded language, as we elicited before, does not lend itself nicely to optimization. This is especially true, when the domain specific entities like pattern definitions for example, are modeled as functions. In that case, it is not possible to analyze the function and optimize it. The alternative to this approach is, model them as algebraic data types; in that case, the interpreter can analyze the definitions and transform it. This approach has been taken in [10] for optimization of their image synthesizing language called PAN. But we advocate that, with this approach, the resulting language is not an embedded language in true spirit as we do not use the compiler *of* the host language, rather we write a compiler *in* the host language.

Our solution is, to take a middle path i.e. we do not model the pattern definitions as data types, but to analyze the function we pass a special argument to pattern functions. When a pattern definition receives this argument, instead of producing pattern instances, it produces some information that is useful for optimization like indices of constraints. This is achieved by overloading the functions.

The limitation of our approach and also that of [10] is, in the resulting language, the user does not have unrestricted access to all library functions, unless the embedded compiler supports it(it is overloaded to accept the special argument). We realize that there are some more limitations in this approach like supporting recursive functions and to be able to do more aggressive analysis and optimization of patterns. In future, we plan to use meta-haskell[30] for optimizing CPL. Meta-haskell provides an handle into the internal representation of Haskell program, which is used by the Haskell compiler. With access to that, we can do almost any type of analysis and transformation on our embedded language program, which is also a valid Haskell program.

```
tTransformer = foldTree0 f ([],0) g
  where f node (acc,level) = (node:acc, level+1)
        g nodes [] = Tree (nodes, [])
        g (_,level) ts = Tree (((fst.label.head) ts,level), ts)

foldTree0 :: (a->c->c) -> c -> (c->[b]->b) -> Tree a -> b
foldTree0 f c g (Tree (a,ts)) =
        let c' = f a c
        in   g c' (map (foldTree0 f c' g) ts)

prune :: (a->Bool) -> Tree a -> [Tree a]
prune p = foldTree0 const undefined g
  where g node ts
           | not (p node) = []
           | null ts      = [Tree (node,[])]
           | otherwise    =
                   let ts' = concat ts
                   in if null ts' then [] else [Tree (node,ts')]

check cons indx dummy = foldTree0 const undefined  g
   where g (ass,level) ts = Tree ((cc (reverse ass) level), ts)
         cc path level    =
               let  inds        = lookup' level indx
                    bools       = map (cons patt!!) inds
                    patt        = toPatt path dummy
               in not (elem False bools)
```

Figure 4.3: Haskell Code for Backtracking

Chapter 5

Lazy Divide and Conquer Constraint Solving

In this chapter, we present the constraint solving algorithm that is used in CPL implementation.

5.1 Background

We formulate constraint satisfaction problems as constraint networks. A *constraint network* consists of a set of *variables* $X = \{x_1, \ldots, x_n\}$, *domains* $D = \{D_1, \ldots, D_n\}$, $D_i = \{v_1, \ldots, v_m\}$ (finite domain) and *constraints* $R_{S_1}, \ldots, R_{S_k}$, where $S_i \subseteq X$, $1 \leq i \leq k$. For any subset of variables $S = \{x_1, \ldots, x_r\}$, the *domain* of S is $D_1 \times \ldots \times D_r$. A *constraint* is a pair (R, S) where S is a subset of variables $S \subseteq X$, also called its *scope* and R is a relation defined over the domain of S, whose tuples denote the legal combination of domain values. The pair (R, S) is also denoted R_S. The *arity* of a constraint R_S is defined as the cardinality of S.

Given a set of variables $Y \subseteq X$, we define $\mathcal{K}_Y$ as follows:

$$\mathcal{K}_Y = \bigcup\{R_{S_i} \mid S_i \subseteq Y, 1 \leq i \leq k\}.$$

As an example, an n-queen problem can be formulated as a constraint network consisting of a set of variables $X = \{q_1, \ldots, q_4\}$, domains $D = \{D_1, \ldots, D_4\}$ such that $D_i = \{1, 2, 3, 4\}$, for $1 \leq i \leq 4$, and constraints $\bigcup_{1 \leq i < j \leq 4} R_{\{q_i, q_j\}}$, where $R_{\{q_i, q_j\}}$ specifies that a queen at row i cannot attack another queen at row j. Furthermore, $\mathcal{K}_{\{q_1, q_2, q_3\}} = R_{\{q_1, q_2\}} \cup R_{\{q_1, q_3\}} \cup R_{\{q_2, q_3\}}$. $\mathcal{K}_{\{q_1, q_2, q_3\}}$ represent the set of all constraints involving q_1, q_2 and q_3.

5.2 Motivation for Lazy D&C

In this section, we give the motivation for using lazy d&c strategy through simple examples. Consider Figure 5.1, which shows the partial search space for 4-queen problem. There are 4 variables in the problem: q_1, q_2, q_3 and q_4, each represents a level in the search space, and the first level (the root) contains a dummy node. A node in the tree at level i with value v represents the assignment $q_i = v$.

A solution is represented by a complete path in the corresponding solution tree, joined by solid edges. A complete path is one that contains assignment to all variables in the subproblem. The only complete path shown in the figure contains the assignment $q_1 = 3$, $q_2 = 1$, $q_3 = 4$, and $q_4 = 2$. Moreover, a dashed edge joined a node at level i to another at level $i + 1$ represents an assignment of a value to the variable at level $i + 1$ that fails a consistency check. Consequently, no further search is carried out starting from the node of each dashed edge.

Two nodes labeled a and b in the tree have value 1, indicating that the 2nd queen (q_2) is positioned at column 1. The search spaces rooted at a and b are identical, as both of them are associated with three constraints namely, $R_{\{q_2, q_3\}}$, $R_{\{q_2, q_4\}}$, and $R_{\{q_3, q_4\}}$, with q_2 being assigned the value 1. Comparing the subtrees rooted at a and b, we see that two dashed edges, joining pair of assignments $(q_2 = 1, q_3 = 1)$ and $(q_2 = 1, q_3 = 2)$ respectively, are explored and discarded at both subtrees. This re-computation can be avoided if we can save our exploration result at a and reuse it at b.

However, backtracking algorithm, or any of its intelligent siblings like back-jumping or backmarking [14], do not achieve the desired saving. Backmarking comes close to meeting our expectation; it does not redundantly check the constraints involving a node living in the subtree and the common ancestors of nodes a and b. However, in this example, backmarking does not eliminate redundant checks, as the common ancestor of nodes a and b is just the dummy root node.

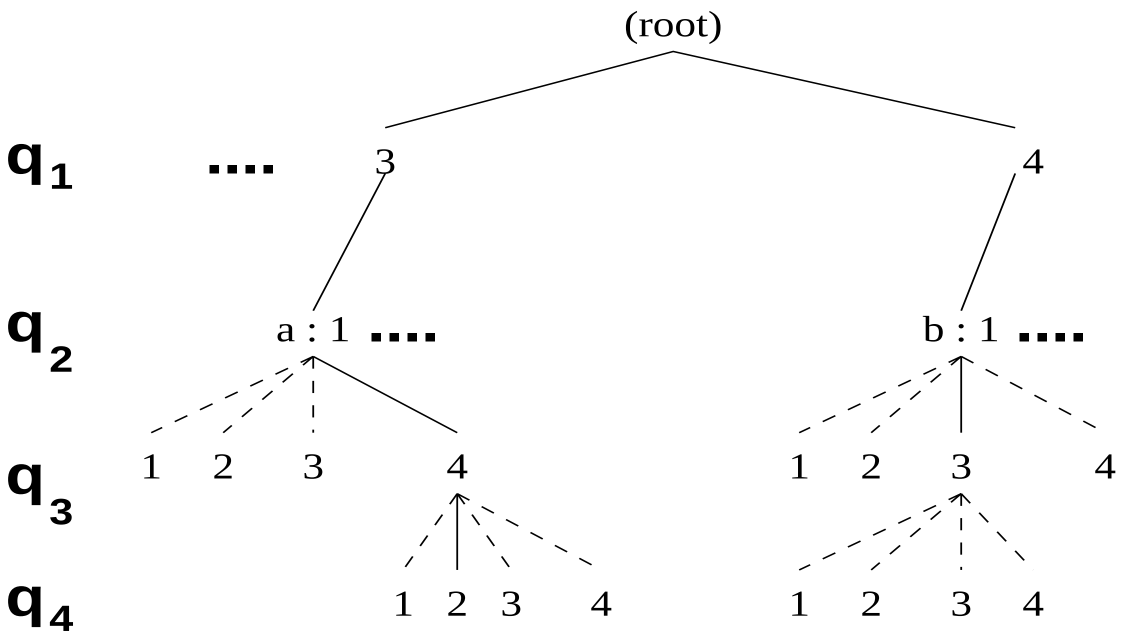

Figure 5.1: Partial Solution Trees of 4-queen problem

In general, caching a solution subtree for future reuse can reduce the number of consistency checks drastically, since the number of identical nodes(nodes with same assignments) occurring at a level can be potentially exponential, resulting in exponential number of identical subtrees. Every time a solver comes to such a node, like a in level q_2 of Figure 5.1, it has to check the constraints involving variables that lie in or below level 2 of the tree. In the worst case, the complete subtree might get re-evaluated.

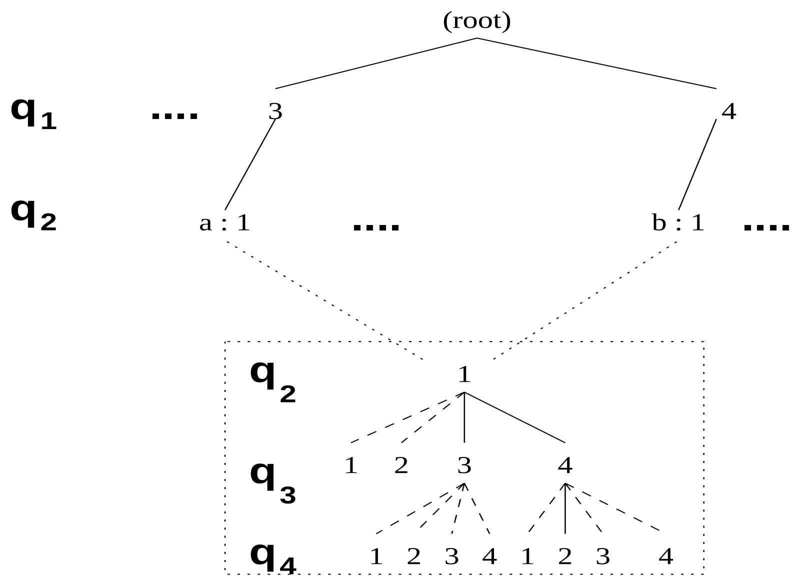

Figure 5.2: Partial Solution Tree of 4-queens problem divided into two subproblems with variables q_1, q_2 and q_2, q_3, q_4.

We can view the subtree rooted at a as representing the search space for a variant of 3-queen problem (with domain values ranged from 1 to 4). In the spirit of divide-and-conquer strategy, we wish to solve this subproblem and combine its result with those obtained from other subproblems. In many traditional problems, such as sorting, we can intuitively divide a problem into independent subproblems, and merge the individual results to form the final solution. For constraint satisfaction problems, however, it is generally not possible to cleanly divide a problem into subproblems that can be solved independently. For instance, if we divide the 4-queen problem such that q_1 and q_3 live in different subproblems, there is a constraint, $R_{\{q_1,q_3\}}$, connecting these two subproblems. So when should such constraint be checked?

One simple solution, is to first ignore those constraints that involve variables living in different subproblems, solve each of the subproblems and later use the ignored constraints to filter their combined solution. But this forceful division of problem may cause redundant consistency checks; some instantiations of the solution of one subproblem, Π_1 say, may prune part of the search tree of another subproblem Π_2, and thus make the computation of that part of Π_2's solution redundant.

Figure 5.2 shows a division of 4-queen problem into two subproblems: The first subproblem contains variables $\{q_1, q_2\}$, and the second contains variables $\{q_2, q_3, q_4\}$. (The astute readers will notice the repetition of variable q_2 in both subproblems. This arrangement will facilitate merging of solutions from two subproblems (*cf.*, Section 5.3.2).)

When solving the second subproblem independently, two subtrees rooted at level q_3, with assigned values 3 and 4 respectively, are explored. However, as we have already seen in Figure 5.1, if we were to choose only one solution for the entire problem, we would only need to explore the path passing through node a, and explore the subtree rooted at the node with assignment $q_3 = 4$. In this case, the computation performed at second subproblem on subtree rooted at $q_3 = 3$ becomes redundant.

In general, consistency checks done at subproblem can become redundant even when we are finding all solutions to a CSP. This problem can be addressed conveniently by a lazy computation framework like that of Haskell [28]. Subproblems are solved lazily, whenever they are demanded. The demand arises when constraints connecting subproblems are checked to produce final solutions for the entire problem. This demand-driven nature of computation allows some parts of the search trees of subproblems to remain unevaluated, thus eliminating the risk of redundant consistency checks.

5.3 Algorithm

We identify a constraint satisfaction problem with a constraint network, and a subproblem with a subnetwork. The lazy divide-&-conquer algorithm is defined by function $DivCon_S$, as shown in Figure 5.3. The function is subscripted by a solver S, and takes in two arguments: a constraint network N; and a map p describing how to divide N into subnetworks. If the network is not to be divided (*ie.*, the map p is a primitive map), the solver S is called to find all solutions for N. Otherwise, function *divide* is called to partition the network into a list of subnetworks; it returns two components: *nms* is the list of subnetworks and their associated "submaps" describing how each subnetwork should be further divided. C is the set of global constraints (taken from N) inter-connecting these subnetworks. Each subnetwork is solved recursively. The results are combined, and the global constraints are solved (with the same solver S) by calling the function $join_S$.

```
function $DivCon_S$ ($N$ : network, $p$ : map) : solnSet
    local  $\sigma$ : solnSetSet
           $nms$ : network-mapSet
    if $primitiveMap$ ($p$) then return $S(N)$ ;
    $(C, nms) \leftarrow divide$ ($N$, $p$) ;
    for each $(N', p') \in nms$ lazy-do
        add $DivCon_S$ ($N'$, $p'$) to $\sigma$ ;
    return $join_S(C, \sigma)$
```

Figure 5.3: Lazy Divide-&-Conquer Algorithm

The algorithm is executed *lazily*. Firstly, every function call invocation is performed lazily; when a function is applied to a tuple of arguments, the call invocation does *not* begin with the evaluation of its arguments. Rather, the function body is executed first. When the value of an argument is required to advance the execution, that argument is evaluated up to the point when the required value is obtained. For example, if an argument will evaluate to a linear list of values, and only the first two elements of the list are needed to complete the function-body execution, then the argument will be evaluated to produce the first two elements; it will not produce the other list elements. In programming terminology, this suspension of an expression evaluation is achieved by building a *thunk* to encapsulate the computation of the expression. When the value of an expression (in this case, the argument) is required, its corresponding thunk will be *forced*.[1]

In addition to lazy function calls, a lazy-loop command, **for each** ... **lazy-do**, executes its loop-body *lazily* too. Instead of eagerly adding solution set of subproblem (N', p') to σ at each iteration, a thunk corresponding to the execution of this addition (of the solution set) is created. This thunk will be forced when the corresponding solution set is needed (by the $join_S$ operation). Consequently, during run-time, σ does not really contain all solution sets. For those subproblems that have been (partially) solved, σ keeps their (partial) solutions else σ maintains the thunks representing the pending computations.

5.3.1 Map

A map describes how a network can be subdivided. Our strategy assumes some variable ordering. When dividing a network, we assert that subnetworks preserve this variable ordering; *ie.*, variables occurring in a subnetwork must obey the same variable ordering as the original network.

Network division is done such that two consecutive subnetworks share exactly one common variable. This facilitates merging of solution sets of subnetworks at

[1]The thunk must contain the information required to execute the expression. The process of executing the expression in a thunk is called *forcing* [16].

later stage (as will be explained in Section 5.3.2.) Hence, given a network with variable ordering

$$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$$

The division

$$(x_1, x_2, x_3, x_4) \texttt{ and } (x_4, x_5, x_6, x_7, x_8)$$

is valid, but the division

$$(x_1, x_2, x_3, x_4) \texttt{ and } (x_5, x_6, x_7, x_8)$$

is invalid because the first two subnetworks do not share a common variable.

Without loss of generality, we assume that division of a map always produce two submaps, and division operations can be nested.

In defining a constraint-satisfaction problem, we first order the variables, and index them, beginning from 1. Hence, a division of a map can be succinctly described by specifying the index of the common variable. The syntax of a map can be expressed by the following grammar:

$$\text{map} \ ::= \ \text{map} :: \textit{int} :: \ \text{map} \ | \ \text{None}$$

For example, the earlier example of dividing a network of eight variables can be expressed as:

$$p \ = \ \text{None} \ :: \ 4 \ :: \ \text{None}$$

A possible nested map will be:

$$p' \ = \ \text{None} \ :: \ 4 \ :: \ (\text{None} \ :: \ 7 \ :: \ \text{None})$$

Given a network N of eight variables, and the above map p'. The call *divide* (N, p') will return

$$(C_4, \{(N_1, \text{None}), (N_2, (\text{None} \ :: \ 7 \ : \ \text{None}))\})$$

where N_1 consists of the set of variables $A = \{x_1, x_2, x_3, x_4\}$, and constraints $\mathcal{K}_A$; N_2 consists of the set of variables $B = \{x_4, x_5, x_6, x_8\}$, and constraint C_B, where $C_B = \mathcal{K}_B \setminus R_{\{x_4\}}$. Thus, all unary constraints on the common variable x_4 appears in the first subnetwork, but not the second. Lastly, $C_4 = \mathcal{K}_{A \cup B} \setminus (\mathcal{K}_A \cup \mathcal{K}_B)$ is the set of constraints between variables in (A - $\{x_4\}$) and (B - $\{x_4\}$); *ie.*, the inter-connecting constraints between N_1 and N_2.

To proceed further, $\mathit{divide}(N_2, (\text{None} \ :: \ 7 \ :: \ \text{None}))$ yields

$$(C_7, \{(N_{21}, \text{None}), (N_{22}, \text{None})\})$$

where N_{21} consists of variables $D = \{x_5, x_6, x_7\}$, and constraints $\mathcal{K}_D$; N_{22} has variables $E = \{x_7, x_8\}$, and constraints C_E, where $C_E = \mathcal{K}_E \setminus R_{\{x_7\}}$. Lastly, $C_7 = \mathcal{K}_{D \cup E} \setminus (\mathcal{K}_D \cup \mathcal{K}_E)$.

Note that unary constraint on the common variable appears only in one subnetwork. If it were to exist in both subnetworks, redundant consistency checks of this unary constraint would occur. While we shall not delve into the technical details, we claim that placing this unary constraint in the second subnetwork will also incur redundant checks, and thus the decision to place it with the first subnetwork.

5.3.2 Function $join_S$

Function $join_S$ combines the solution sets of all subproblems created earlier by the *divide* function, and subjects them to global constraint (C) satisfaction. Because of the existence of common variable between two subproblems, combining two solution sets is equivalent to "equi-join" operation on two tables in database, where two tables are joined on the condition of equality of their common attribute (the common variable, in our case.)

Under lazy evaluation, when the solution set of a subproblem is needed for the first time, its corresponding thunk is forced, resulting in the thunk being replaced by the (progressively constructed) solution set.

Demand for the solution set of a subnetwork arises from the need to solve global constraints C (which is needed in order to construct a solution for the entire constraint network.) In solving global constraints, we can employ techniques different from that provided by the solver S, although it is simpler to use the same solving technique throughout the entire process.

The result of $join_S$ will be one (possibly empty) solution set that satisfies *all* constraints on the variables of the combined subproblems.

5.3.3 Solving $S(N)$

In the degenerate case (with the presence of a primitive map $None$), the call $S(N)$ is also executed lazily. Note that $S(N)$ yields solutions that only satisfy constraints local to the subproblem, and these solutions may not satisfy constraints interconnecting the current subproblem with other subproblems. The possible output of $S(N)$ are as follows:

1. $S(N)$ may not be executed at all, if there is no need for its solution set. This happens when the algorithm aborts because one subproblem (not the current N) turns out to be unsatisfiable.

2. $S(N)$ may be executed to produce all solutions. This happens when the algorithm is used to generate all possible solutions to the constraint-satisfaction problem.

3. $S(N)$ may be executed to produce some (and possibly all) solutions. This happens when the algorithm is used to generate just one solution, and the execution of $S(N)$ will halt when the first solution that satisfies the global constraints connecting the subproblems is found.

It is important to understand that, due to lazy evaluation, all functions are executed in an interweaving manner. Thus, it is not the case that $S(N)$ runs to its completion and passes its result to $join_S$. Rather, we can imagine both $S(N)$ and $join_S$ as running in a fine-scale co-routine manner, with $join_S$ pauses to request for some intermediate result from $S(N)$, and continues when the requested result is available.

5.4 Experimentation

We have implemented our lazy D&C strategy in Haskell – the *de-facto* lazy functional language. In our implementation, the subproblems corresponding to primitive map are solved using backtracking. Given two subproblems sharing a common variable, the $join_S$ operation solves left subproblem first and then the second subproblem(lazily) and checks the global constraints in backtracking manner on the combined solution.

To illustrate the effectiveness of our approach, we apply our program to solve n-queen problems. We benchmark it against the standard search algorithms developed by Nordin *etc.* in Haskell [25]. A widely used metric for evaluating constraint solving algorithms is the number of consistency checks performed by the algorithms. Adopting this metric, we obtain very encouraging results, as shown in Table 1. Notice that in all cases, our d&c strategy performs the least number of consistency checks.

In the following, we present two experiments. All measurements are taken on a 1.7Ghz pentium IV machine running linux. The programs were compiled with ghc-5.02 with -O2 and profiling flags.

Table 5.2 shows the effect of different partitioning on the number of consistency checks and running time. Here, each test case represents a different map for 12-queen problem and `DivCon`$_{\texttt{bt}}$ is run using the map. For simplicity, we express the map in list format: a list `[a,b,c]` is equivalent to the map `None :: a :: (None :: b :: (None :: c :: None))`.From the table, we can see that the

Queens	8	9	10	11	12
`bt`	46752	243009	1297558	7416541	45396914
`bjbt`	41128	214510	1099796	6129447	36890689
`bjff1`	11579	47375	191776	868066	4280093
$\texttt{DivCon}_{\texttt{bt}}$	10064	40548	172198	790634	3976570
Solutions	92	352	724	2680	14200

Legend:		
	`bt`	backtracking algorithm [25]
	`bjbt`	backjumping algorithm [25]
	`bjff1`	backjumping with forward checking
		(The fastest algorithm reported in [25])
	$\texttt{DivCon}_{\texttt{bt}}$	d&c framework with backtracking solver

Table 5.1: Number of consistency checks performed by various algorithms on the all solutions to n-queen problem.

number of consistency checks decreases with finer granularity of partitioning.

This experiment shows that $\texttt{DivCon}_{\texttt{bt}}$ solves the n-queen problem the best when the map is right-skewed, and has finest granularity. We contribute this behavior to the following two reasons: 1. The n-queen problem has uniform distribution of constraints over all variables; 2. The solver `bt` and the operation $join_{\texttt{bt}}$ find solutions in depth-first manner, traversing the list of variables from left to right. Thus, maximal reuse of solutions can be attained by grouping the subproblem in right-associative fashion. But in general, topology of constraint network , the solver used to solve the subproblems and the method of combining the solutions of subproblems would decide the goodness of a particular division of the problem.

Test case	Consistency checks	Time (seconds)
`[ ]`	45,396,914	23.1
`[2]`	28,398,804	20.8
`[2,3]`	18,087,264	22.5
`[2,3,4,5]`	7,978,112	33.2
`[2,3,4,5,6, 7,8,9,10,11]`	3,976,579	66.7

Table 5.2: Impact of granularity on running time for 12-queen problem

Table 5.3 shows the effect of different partitioning on the heap usage and number of consistency checks. Heap usage is measured in MBytes X Seconds. If size of the live heap is plotted against time, heap usage is measured by the area enclosed between the time axis and the live heap curve over the total running time of the program. The figures demonstrate that with increased granularity heap usage increases and number of consistency checks decreases. Greater heap usage is due to the fact that with increase in granularity solutions of greater number of subproblems are saved and reused.

5.5 Application to CPL

In this section we demonstrate the usefulness of our d-&-c framework for efficient pattern matching in CPL.

As mentioned before, head-and-should is one of the most popular patterns used

Test case	Consistency checks	Heap Usage (MBytes X seconds)
[ ]	1,297,558	0.001
[2,3]	545,392	10.01
[2,3,4,5]	262,822	33.17
[2,3,4,5,6,7	178,816	48.52
[2,3,4,5, 6,7,8,9]	172,198	52.16

Table 5.3: Impact of granularity on heap usage for 10-queen problem

by analysts–it bodes of decline of price. Appendix A gives one definition of head-and-shoulder pattern in CPL. But to show the effectiveness of d-&-c approach, below we provide three different definitions of the pattern; they differ in the way the pattern is divided. The definitions uses simple Haskell syntax.

```
global = (up >.> down >.> up >.> down >.> up >.> down) ? [..]
conc   = let b2d = (down >.> up) ? [..]
             b2e = (b2d >.> down) ? [..]
             b2f = (b2e >.> up) ? [..]
             b2g = (b2f >.> down) ? [..]

         in (up >< b2g) ? [..]
right  = let e2g = (up >.> down) ? [..]
             d2g = (down >.> e2g) ? [..]
             c2g = (up >.> d2g) ? [..]
```

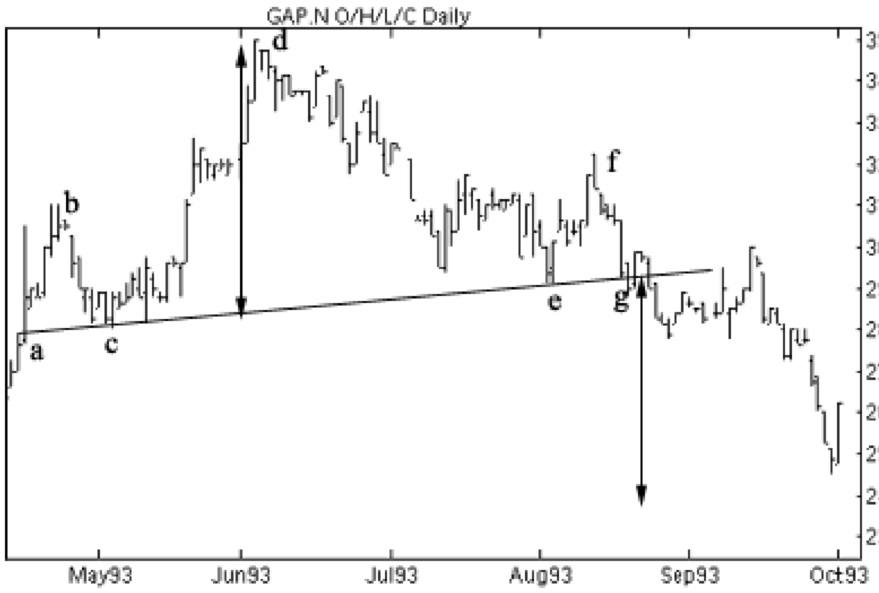

Figure 5.4: Head and Shoulder

```
          b2g = (down >.> c2g) ? [..]
      in  (up >.> b2g) ? [..]
```

n the definition of `global`, all constraints are globally specified; the pattern thus has the coarsest granularity. In `conc`, the sub-patterns are composed in left-associative manner. In the equation `b2e = (b2d >.> down) ?  [..]`, the list of constraints (`[..]`) contains global constraints connecting sub-patterns `b2d` and `down` (shown in Appendix A). Lastly, in `right`, sub-patterns are composed in right-associative manner. Both `conc` and `right` has the finest granularity.

Table 5.4 shows the time taken to search all instances of head-and-shoulder pattern over 20 years of daily data of 20 companies, under different decompositions and maximum lengths of patterns. In practice, patterns that occur over a very long period of time are not useful for forecasting. So the program only searches for patterns of length less than a particular value given by the user.

Division Variation	Maximum Length of a pattern (days)			
	250	450	650	850
`global`	16.5	24.4	29.8	33.8
`conc`	14.0	17.2	20.6	23.0
`right`	12.3	15.2	17.0	19.0
Number of instances found	21	26	27	27

Table 5.4: Running time (in seconds) for searching all head-&-shoulder pattern instances

Matching a pattern defined by `global` is similar to solving a constraint network using backtracking algorithm. While backtracking algorithm is still used at subproblems, we see that pattern matching can be done efficiently using lazy d&c strategy with finer granularity. The motivation behind decomposing a pattern into `conc` format is that it is efficient to have subgraphs with more local constraints, so that there will be more saving obtained from reusing its solutions. For the current pattern, we find that there are most constraints associated with landmarks `b` to `f`. Thus, we chose to make that a `b2f` a subproblem, and provide further division of this subproblem. As one can see, there can be many way to partition a problem, and the topology of the constraint graph can be exploited during partitioning. However, unlike other works, where decomposition is possible only with a topology that satisfy some properties, our algorithm works for any arbitrary decomposition; though performance may be different.

It is interesting to see that the essence of our lazy d&c strategy has been encoded in the operators `>.>` and `?` of CPL. As such, a pattern defined using these operators reflects actual division of a pattern. Such encoding thus opens up the opportunity for a CPL system to transform and optimize the execution of the lazy d&c strategy, through transformation of pattern definition at compile time.

5.6 Related Works & Future Work

The prevalent method used in decomposing constraint network is the tree clustering method [8, 15]. The objective has been to identify constraint networks that can be compiled into an equivalent tree of subnetworks whose respective solutions can be combined into a complete solution efficiently. Some of these tree-decomposition schemes are join-tree clustering [8], junction-tree decompositions, and hyper-tree decomposition [15]. In contrast, what we propose in this chapter is a scheme for programmer to freely divide the network, through introduction of maps, in a manner that fits the properties of the network. To support general division of network, we show that lazy evaluation is crucial to the elimination of redundant consistency checks. It will be interesting to determine if the introduction of lazy evaluation can help broaden the range of constraint networks handled by the existing tree-decomposition schemes.

Lazy evaluation is not new to constraint satisfaction algorithms. A flavor of this technique was first exploited by Zweben and Eskey in [33] and later in [9] and [29] to improve forward checking and arc consistency algorithms. Although these algorithms delay some constraint checks, they do not really require a lazy-evaluation framework. In a stroke, these algorithms cleverly infer the amount of "laziness" required to avoid redundant checks[2], so it is more apt to describe these algorithms as "minimal" rather than "lazy". Our algorithm addresses the redundant computation problem associated with application of divide-and-conquer approach to

[2]Bacchus and Grove remarked that "we rediscovered this (minimal forward checking) not by thinking about lazy evaluation, but instead as a corollary of our results connecting backmarking and forward checking" in page 3 of [5].

constraint satisfaction problem by employing lazy evaluation. We show that, by performing backtracking algorithm in a lazy d&c framework, we can eliminate redundant computations, leading to a reduction in consistency checks. But this improvement comes with a price; the space complexity of the algorithm is too high for the algorithm to scale up to large problems. One solution is related to "learning" algorithm[12] for constraint solving, where as the algorithm explores different frontiers of the search space, it learns from it mistakes, that is, when it reaches a node in the solution tree that can not be further extended, it records "nogoods" that will prevent it to come to the same dead end again. A nogood is a set of assignments that is not a subset of any solution of the problem. The nogoods recorded during learning represents an approximation of the already explored search space. In our current algorithm, we store a complete copy of the search space explored so far, thus increasing the space requirement. But we can instead store nogoods as an approximation of the search space like learning algorithms.

Chapter 6

Conclusion

In this thesis, we have introduced a very interesting application of programming language in finance and business. Forecasting future trend of the stock market is a long researched area. Probably the attractiveness of the end result has drawn interest from multiple disciplines of science to this problem. And the author was no exception; whom the challenge of using computer science to investing drew him to this problem.

There are many approaches to stock forecasting problem, ranging from application of esoteric techniques of artificial intelligence area to fundamental and technical analysis. Later techniques are more accepted in the actual practice. Our topic comes from technical analysis of stock, which interpret the patterns in stock market price charts as precursor to future trend of the market. So the technical analysts have a need to program their patterns and search them in the huge database. To this effect we have designed and implemented a domain specific language to program chart patterns.

We chose to embed our language in a state-of-the-art functional language Haskell instead writing a compiler for our language. The decision was motivated by the success of Haskell in hosting a flurry of small domain specific languages in recent past. Some of the salient features of our language are,

1. It provides a high-level platform upon which analysts can define and search patterns easily without much programming expertise; this is different from works in neural network which treat the pattern discovery process as a black box [20] – something that the user can be uncomfortable with.

2. Its specification is close to how analysts would describe the patterns in human terms(fuzzy constraints), but without compromising on precision.

3. Patterns are defined in a compositional style, which is how the real analysts like to define.

4. It allows to define new technical indicators and use them in pattern definitions to specify constraints. This allows an easy and elegant way to specify constraints and reuse them.

Embedding domain-specific languages within a functional language does not only provide the domain experts with elegant and convenient specification of domain problem, but also brings about many benefits from the host language. But efficiency of embedded languages often turn out to be unsatisfactory. Because, any attempt to apply domain specific optimizations is hindered by the inability to inspect the functions, which is invariably needed when the domain specific entities are modeled as functions(most likely case when the host language is Haskell.) The obvious solution to this problem is to write a compiler in the host language. But we chose to take a slightly different approach, which we find interesting and believe to be general enough to be applied to other fields. Our approach relies on Haskell type classes and other advanced features of the type system. Besides this problem, we face few more problems, specific to our application. We take advantage of laziness of Haskell to devise an elegant to solution to it. Though by tricky manipulation of type system and laziness, we have been successful in incorporating some domain specific optimizations into our system, we realize that writing a compiler for our language is inevitable to be able to do more powerful analysis and transformations.

In course of implementation, we have discovered a new approach to constraint solving based on divide and conquer. we identify the redundant computation problem associated with application of divide-and-conquer approach to constraint satisfaction problem. We propose a novel way of overcoming this problem by employing lazy evaluation. Our initial experimentation with N-queen problem using this approach has been encouraging and enlightening. Enlightening, because it led us to many problems, that we did not realize before. Most importantly, the space complexity of our algorithm is prohibitively large to be able to scale upto large problems. Though we still believe that the approach is novel. Lately, we have adopted our approach into no-good driven learning algorithms for constraint solving and the results are encouraging. We plan to carry on this work further.

Appendix A

Head-and-Shoulder Pattern Definition

The following pattern is used to generate the performance statistics as presented in the paper. The unary operator `cast` converts an integer to a floating-point.

```
h_s  =
   up >.>  down >.>  up >.>  down >.>  up >.> down ?
    \p->
     let  [a,b,c,d,e,f,g] = lms p
          x1 = c
          y1 = low x1
          x2 = e
          y2 = low x2
          z1 = y1+(y2-y1)*cast(g-x1)/cast(x2-x1)
          z2 = y1+(y2-y1)*cast(a-x1)/cast(x2-x1)
          z3 = y1+(y2-y1)*cast(b-x1)/cast(x2-x1)
          h1 = high b - z3
          z4 = y1+(y2-y1)*cast(f-x1)/cast(x2-x1)
          h2 = high f - z4
          z5 = y1+(y2-y1)*cast(d-x1)/cast(x2-x1)
    h3 = high d - z5
    m  = h2 / h3
    n  = h1/h3
in [
    high d > high b,
    high e < high b,
    high d > high f,
    high c < high f,
    close g < z1 && close (g-1) > z1 &&
    low a < z2 && low (a+1) > z2 &&
    abs(2*(h1-h2)/(h1+h2)) < 0.1 &&
    m < 0.7 && m > 0.4 && n > 0.4 && n < 0.7
   ]
```

Bibliography

[1] S.B. Achelis. *Technical Analysis A to Z.* McGraw-Hill Professional, 1982.

[2] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zaït. Querying shapes of histories. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 502–514. Morgan Kaufmann, 1995.

[3] J. Allen. Towards A General Theory of Action and Time. In *Artificial Intelligence*, volume 23, pages 123–154, 1984.

[4] S. Anand, W.N. Chin, and S.C Khoo. A Lazy Divide & Conquer Approach to Constraint Solving. In *International Conference in Tools with Artificial Intelligence(ICTAI)*. IEEE, 2002.

[5] F. Bacchus and A. Grove. On the Forward Checking Algorithm. In *Principles and Practices of Constraint Programming(CP)*, pages 292–308, 1995.

[6] N. Bulkowski. *Encyclopedia of Chart Patterns.* John Wiley and Sons, 1982.

[7] K. Cheng and M. Spetch. Mechanisms of landmark use in mammals and birds. *Spatial Representation in Animals*, 1998.

[8] R. Dechter and J. Pearl. Tree Clustering for Constraint Networkssome Applications of Graph Bandwidth to Constraint Satisfaction Problems. In *Artificial Intelligence*, volume 38, pages 353–366, 1989.

[9] M.J. Dent and R.E. Mercer. Minimal Forward Checking. In *6th IEEE International conference on Tools with Artificial(ICTAI)*, pages 432–438, 1994.

[10] C. Elliot, S. Finne, and O. de Moor. Compiling Embedded Languages. In *Semantics, Applications and Implementation of Program Generation (SAIG)*. Springer Lecture Notes in Computer Science, 2000.

[11] C. Elliot and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, Amsterdam, Holland, August 1997. ACM Press.

[12] D. Frost and Rina Dechter. Dead-end Driven Learning. In *AAAI*, pages 294–300, 1994.

[13] K.S. Fu. *Syntactic Pattern Recognition and Applications.* Englewood Cliffs, 1982.

[14] J. Gaschnig. A General Backtrack Algorithm that eliminates most redundant tests. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1977.

[15] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural csp Decomposition Methods. In *International Joint Conference on Artificial Intelligence(IJCAI)*, pages 394–399, 1999.

[16] Abelson H., Sussman G.J., and Sussman J. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1993.

[18] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Survey*, 28(4es), December 1996. Article No. 196.

[19] M. Humphreys, J. Wiles, and S. Dennis. Towards a theory of human memory: Data structures and access processes. *Behavioral and Brain Sciences*, 17(4), 1994.

[20] K. Kamijo and T. Tanigawa. Stock price pattern recognition - a recurrent neural network approach. In *1990 International Joint Conference on Neural Networks*, pages I215–222, 1990.

[21] S. Kamin. Standard ML as a Meta-Programming Language. Technical report, University of Illinois at Urbana-Champaign, September 1996.

[22] G.J. Klir and T.A. Folger. *Fuzzy sets, uncertainty, and information.* Prentice-Hall, 1992.

[23] Gary Meehan and Mike Joy. Animated fuzzy logic. *Journal of Functional Programming*, 8(5):503–525, September 1998.

[24] S. B. Navathe and Ahmed R. *TSQL-A Language Interface for History Databases.* 1987.

[25] T. Nordin and A. Tolmach. Modular Lazy Search for Constraint Satisfaction Problems. *Journal of Functional Programming*, 11(5):557–587, September 2001.

[26] C. Perng, H. Wang, S.R. Zhang, and S. Parker. Landmarks: A new model for similarity-based pattern querying in time series databases. In *Proc. 16th International Conference on Data Engineering.,2000*, pages 33–42, 2000.

[27] S.L. Peyton Jones, J. Eber, and J. Seward. Composing contracts: An adventure in financial engineering. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 280–292, Montreal, Canada, September 2000. ACM Press.

[28] S.L. Peyton Jones, R.J.M. Hughes, L. Augustsson, D. Barton, B. Bontel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M.P. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P.L. Wadler. Report on the programming language Haskell 98. Technical report, February 1998.

[29] T. Schiex, J-C. Rgin, C. Gaspin, and G. Verfaillie. Lazy Arc Consistency. In *AAAI/IAAI*, volume 1, pages 216–221, 1996.

[30] T. Sheard and S. L. Peyton-Jones. Template Meta-programming for Haskell. In *Haskell Workshop*, 2002.

[31] Richard Snodgrass. The Temporal Query Language tquel. In *ACM Transactions on Database Systems*, volume 12, pages 247–298, 1987.

[32] M. Wallace and C. Runciman. Heap Compression and Binary I/O in Haskell. In *2nd ACM SIGPLAN Workshop on Haskell*, Amsterdam, June 1997. ACM Press.

[33] M. Zweben and M. Eskey. Constraint Satisfaction with Delayed Evaluation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 875–880, 1989.