THE UNIVERSITY OF CALGARY

Pattern Matching In Charity

by

Charles E. Tuckey

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 1997

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "**Pattern Matching In Charity**" submitted by Charles E. Tuckey in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Robin Cockett
Department of Computer Science

Dr. Wayne Eberly
Department of Computer Science

Dr. Claude Laflamme
Department of Mathematics & Statistics

Date

Abstract

Charity, a functional style programming language based in the mathematics of category theory, was developed at the University of Calgary by the **charity** group under the direction of Dr. Cockett. Key to the usefulness of such languages is the ability to define functions using pattern matching. This thesis describes an extended type theory for a **charity** term logic with sophisticated pattern matching capabilities. It gives a translation of this extended term logic to **charity**'s core term logic, enabling programs written in the extended term logic to be evaluated. The translation closely mirrors the type theory.

In addition, this work introduces a new pattern matching feature which is not found in other functional style languages. It shows how the facility of guarding patterns with boolean functions, available in many functional languages, can be extended so that functions of arbitrary types can be used as pattern guards.

Acknowledgements

The work described in this thesis would not have been possible without the help of the **Charity** group at the University of Calgary. First and foremost of the group is Dr. Robin Cockett. **Charity** originated with him and it is certainly his enthusiasm and leadership that has kept the group going over the last five years. Peter Veseley, another member of the **charity** group, developed the first patterned term logic type theory. Without this work having been done the presentation of this thesis would have suffered greatly. Peter also designed and implemented a term logic type checker that made the process of translating patterned term logic to core term logic easier and more straightforward. Group member, Barry Yee, played a large part in the design and implementation of the C version of **charity**. It was this version of **charity** into which the pattern matching discussed in this work was implemented. Marc Schroeder did the design and first implementation of higher order pattern matching. Finally, there is Tom Fukushima, who wrote the original SML version of **charity** which received the first implementation of pattern matching and also served as the structural basis for the current C version of **charity**.

On the personal side, this work would not have been accomplished without the unconditional and loving support of my wife, Judy, and my parents, Joan and Kenn. In addition, there were a large number people who knew I was attempting to write a thesis and, even though the product was three years in the making, they never gave up inquiring if the thesis was done yet. Their interest provided much incentive to finish just to avoid the embarrassment of a public failure.

Finally, I would never have undertaken this project without the financial support provided by an NSERC PSGA scholarship and a Wilfrid R. May scholarship administered by the Heritage Trust Fund Scholarship Foundation.

This document was typeset in LaTeX using Marc Schroeder's thesis style.

Dedication

To Judy

Table of Contents

List of Figures

Chapter 1

Introduction to Charity and Patterns

This work defines a patterned term logic for the **charity** programming language and gives a translation from the patterned term logic to **charity**'s core term logic. The patterned term logic extends **charity**'s core term logic to allow the use of a richer set of patterns in **charity** programs. Patterns are a key feature of functional style programming languages; they allow more succinct expression of programs. For example, the **charity** program for testing the first elements of two lists for equality written without patterns would be:

$$\left\{ \begin{array}{lcl} nil & \mapsto & \left\{ \begin{array}{lcl} nil & \mapsto & true \\ cons\,\_ & \mapsto & false \end{array} \right\} p_1(L, L') \\ cons\,a & \mapsto & \left\{ \; a' \mapsto \left\{ \begin{array}{l} nil \quad \mapsto false \\ cons\,b \mapsto \left\{ \; b' \mapsto eq(a', b') \; \right\} p_0 b \end{array} \right\} p_1(L, L') \; \right\} p_0 a \end{array} \right\} p_0(L, L')$$

Written with patterns the program is shorter and easier to understand:

$$\left\{ \begin{array}{lcl} (nil, nil) & \mapsto & true \\ (cons(a, \_), cons(b, \_)) & \mapsto & eq(a, b) \\ \_ & \mapsto & false \end{array} \right\} (L, L')$$

This opening chapter will introduce the **charity** programming language, motivate the use of extended pattern matching and describe the general characteristics of patterns and pattern matching. Chapter 2 will introduce **charity**'s patterns. Chapters 3 and 4 develop a patterned term logic type theory for **charity**. Chapter 5 gives a translation of the patterned term logic to core term logic. The patterned term logic, all extensions to this term logic described in the thesis and the translation to core term logic have been implemented in a working version of the **charity** programming language.

1.1 Introduction to charity

Charity is a categorical programming language that borrows heavily from the functional programming paradigm. It is based in the mathematics of category theory, which makes it unique. The functional character of **charity** is displayed in its exclusive use of functions and datatypes for information processing, its use of functional composition for building programs, its polymorphic datatypes and its strong typing. These characteristics mean **charity** inherits the benefits of functional programming languages.

Charity's categorical heritage makes it strongly normalizing (guaranteed to terminate). Thus the program verifier does not need to worry about proofs of termination. **Charity** prevents the use of general recursion. This is a major difference from most other programming languages but not using general recursion is "possible theoretically, pragmatically, and practically" (Cockett & Fukushima, 1992).

Charity supports both initial and final datatypes which use constructors and destructors respectively. The datatypes are manipulated in a user term logic which supports the case and fold expressions (for initial datatypes), record and unfold expressions (for final datatypes), and maps (for both). An expression written in the user term logic is evaluated by passing it through a succession of translations. The parser translates the user term logic to the patterned term logic, the subject of this thesis. The patterned term logic is in turn translated into core term logic. This is then translated into categorical combinators which are processed, using rewrite rules, by an abstract machine (Yee, 1995).

The rest of this section gives a brief overview of the datatypes and built in functions of **charity**. More comprehensive information is available from several sources. The theoretical underpinnings of the language are described in (Cockett & Fukushima, 1992), (Cockett & Spencer, 1992a), (Cockett & Spencer, 1992b) and (Schroeder, 1997). The user's manual (Fukushima & Tuckey, 1996) gives a thorough description of the syntax of **charity**. Examples of programs appear in (Cockett, 1992), (Cockett, 1993b), (Cockett, 1993a) and (Schroeder, 1997).

Appendix B lists **charity**'s builtin functions and all datatypes, along with their constructors and destructors, that are used in examples in this thesis.

1.1.1 Datatypes

Charity has two built in types: the integers and characters. These come with relational functions and, in the case of integers, arithmetic functions. The characters are members of the ascii character set.

There are two kinds of datatypes that can be defined in **charity**: the inductive datatypes and the coinductive datatypes. The general datatype definition for inductive datatypes is

$$\text{data } L(A) \to C = \left| \begin{array}{lcl} c_1 & : & E_1(A, C) \to C \\ & \vdots & \\ c_n & : & E_n(A, C) \to C \end{array} \right.$$

where A is taken to be a tuple of type variables and the type of a constructor c_i is gotten by replacing the state variable, C, with $L(A)$; for example,

$$c_n : E_n(A, L(A)) \to L(A).$$

The general datatype definition for coinductive datatypes is

$$\text{data } C \to R(A) = \left| \begin{array}{lcl} d_1 & : & C \to F_1(A,C) \\ & \vdots & \\ d_r & : & C \to F_r(A,C) \\ d_{r+1} & : & C \to E_{r+1}(A) \Rightarrow F_{r+1}(A,C) \\ & \vdots & \\ d_n & : & C \to E_n(A) \Rightarrow F_n(A,C) \end{array} \right.$$

where A is again taken to be a tuple of type variables. Destructors d_i, for $1 \leq i \leq r$, are first order destructors. Destructors d_j, for $r+1 \leq j \leq n$, are higher order destructors. The type of a first order destructor is gotten by replacing the state variable, C, with $R(A)$ as in the example

$$d_r : R(A) \to F_r(A, R(A)) \quad .$$

The type of a higher order destructor is more complex as the destructor is parameterized by the type before the $\Rightarrow$ in the datatype declaration. For example,

$$d_n : E_n(A) \times R(A) \to F_n(A, R(A)) \quad .$$

1.1.2 Built in charity functions

In **charity** all computation is delivered by datatype definitions. The two sorts of datatypes, inductive and coinductive, come with special syntax to deliver computation.

Inductive datatypes

Defining an inductive datatype automatically delivers three functions for operating on the datatype. They are the case, map, and fold. Each of these has its own peculiar syntax which we exemplify below. A precise definition of the syntax and typing of these functions is given in chapter 3.

First, we define a datatype for boolean values:

$$\text{data } \mathit{bool} \to C = \begin{array}{lcl} \mathit{false} & : & 1 \to C \\ \mathit{true} & : & 1 \to C \end{array}$$

The type 1 stands for the unit datatype; there is only member of this type and it is represented by (). The *bool* datatype has been declared with two functions, *true* and *false*, called constructors. Both constructors are maps from the unit datatype to the *bool* datatype.

The above declaration has automatically given us a case function that maps values of type *bool* to some other type. An example of a boolean case function is

$$\left\{ \begin{array}{lcl} \mathit{false}() & \mapsto & \mathit{true}() \\ \mathit{true}() & \mapsto & \mathit{false}() \end{array} \right\}$$

This function maps *false* to *true* and vice versa. The constructors on the left of the case function are patterns. They do not produce values; they serve solely as a way to differentiate input to the function. If the function is applied to a *false*() value then this will match the *false*() pattern and the constructor *true*() will be executed.

Next, we define a "container" datatype that can hold the values of other types:

$$\text{data } \mathit{list}(A) \to C = \begin{array}{lcl} \mathit{nil} & : & 1 \to C \\ \mathit{cons} & : & A \times C \to C \end{array}$$

The $\times$ (product) datatype pairs two values together. A pair can be broken with destructors p_0 and p_1 to get the left value and right value of the pair respectively.

Thus the *cons* constructor takes a pair of values, one of some unspecified type A, the other of type $list(A)$, and maps the pair to $list(A)$. A three element list of boolean types, where A is specified as type *bool* is represented as

$$cons(false(), cons(true(), cons(true(), nil())))$$

Using the *list* datatype we can construct more complex (and useful) case functions. For instance,

$$\left\{\begin{array}{lcr} nil() & \mapsto & 0 \\ cons(true(), cons(true(), nil())) & \mapsto & 2 \\ cons(true(), nil())) & \mapsto & 1 \\ x & \mapsto & -1 \end{array}\right\}$$

The above case function applied to $cons(true(), nil()))$ would yield 1.

In **charity**, the unit element is assumed by default and it is not required to write it explicitly unless it is standing alone. Thus, for example, $false()$ can be more simply written as *false*.

The map function for lists gives a mapping of type variables from one type to another type. Suppose we had a list of boolean values and we wanted to convert it to 0s and 1s, 0 representing *false* and 1 representing *true*. To implement this we would write:

$$list\left\{\begin{array}{lcl} false & \mapsto & 0 \\ true & \mapsto & 1 \end{array}\right\}$$

Charity provides limited recursion for inductive datatypes in the form of the fold function. Again, suppose we had a list of boolean values but this time we wanted to count the number of *true* values in the list.To implement the *true* counter function we would write:

$$\left\{ \begin{array}{l|lll} nil & () & \mapsto & 0 \\ cons & (true, L) & \mapsto & L+1 \\ & (\_, L) & \mapsto & L \end{array} \right\}$$

This function returns a 0 if applied to an empty list. For lists with boolean values, if a value is *true* then the count of *true*s in the earlier part of the list is incremented by 1. If the value is not *true* then the current count is maintained. The _ in the last pattern of the *cons* is a special type of variable called the don't care variable. It is used to represent a value that the programmer does not need.

Other inductive datatypes that will be encountered in this thesis are *nat*, the datatype of unary natural numbers,

$$\text{data } nat \to C = \begin{array}{lll} zero & : & 1 \to C \\ succ & : & C \to C \end{array}$$

SF, the datatype of exceptions,

$$\text{data } SF(A) \to C = \begin{array}{lll} ff & : & 1 \to C \\ ss & : & A \to C \end{array}$$

and *Llist*, the datatype that uses lazy products (see below) to implement lazy lists,

$$\text{data } Llist(A) \to C = \begin{array}{lll} Lnil & : & 1 \to C \\ Lcons & : & Lprod(A, C) \to C \end{array}$$

Coinductive datatypes

Defining a coinductive datatype automatically delivers three functions for operating on the datatype. They are the record, map, and unfold. As with the built in functions for inductive datatypes, we exemplify the syntax below and leave a precise description of the syntax and typing until chapter 3.

Products are eagerly evaluated in **charity**, while all other coinductive datatypes are evaluated lazily. A lazy product can be implemented with the following datatype definition:

$$\text{data } C \to \mathit{Lprod}(A,B) = \begin{array}{lcl} \mathit{fst} & : & C \to A \\ \mathit{snd} & : & C \to B \end{array}$$

A record is used to hold values of a coinductive datatype. The following record gives the elements of a lazy pair:

$$(\mathit{fst}: \mathit{false}, \mathit{snd}: 1)$$

Destructors retrieve values from records. The **charity** term

$$\mathit{snd}(\mathit{fst}: \mathit{false}, \mathit{snd}: 1)$$

would evaluate to 1.

Maps are used with coinductive datatypes in the same manner as with inductive datatypes. The following maps booleans to integers and integers to booleans:

$$\mathit{Lprod}\left\{\begin{array}{lcl} \mathit{false} & \mapsto & 0 \\ \mathit{true} & \mapsto & 1\,, \\ 0 & \mapsto & \mathit{false} \\ \_ & \mapsto & \mathit{true} \end{array}\right\}$$

Note that the *Lprod* map must supply two comma separated functions: one for the A type parameter and one for the B type parameter.

We will explain unfolds using the *inflist* datatype definition below:

$$\text{data } C \to \mathit{inflist}(A) = \begin{array}{lcl} \mathit{head} & : & C \to A \\ \mathit{tail} & : & C \to A \times C \end{array}$$

This datatype can be thought of as representing an infinite list of values of some type A. The values for the list are generated with an unfold. This unfold generates an infinite list of integers:

$$\left(\!\!\left|\; i \;\mapsto\; \begin{array}{lcl} head & : & i \\ tail & : & i+1 \end{array} \right.\right|\!\!\right)$$

An unfold maintains a state that is used to produce a value when the unfold is destructed. If the above unfold was given a 0 as its initial input then, when it was destructed with *head* destructor, it would give 0 as output. However, if it was first destructed with the *tail* destructor before the *head* destructor was applied then it would generate a 1 since destructing with *tail* destructor incremented i, the state, by one.

Higher order destructors apply functions to data. The *stack* datatype is defined below:

$$\text{data } S \to stack(A) = \begin{array}{lcl} pop & : & S \to S \times A \\ push & : & S \to A \Rightarrow S \end{array}$$

The $\Rightarrow$ in the type signature of the *push* destructor indicates that *push* is higher order. The *push* takes as input a term of type S and some input of type A. It then produces output of type S. A type variable which appears on the right of a destructor's type signature is said to be covariant. A type variable that appears on the left of the function mapping of a higher order destructor is said to be contravariant. A type variable that appears in both places, such as A above, is said to be bivariant.

The unfold given below implements a stack which can be emptied (as explained below):

$$\left(\!\!\left|\; \begin{array}{lll} nil & \mapsto & \left| \begin{array}{l} push : a \mapsto [a] \\ pop : ([\,], -1) \end{array} \right. \\ \\ cons(a, l) & \mapsto & \left| \begin{array}{l} push \left| \begin{array}{lcl} -1 & \mapsto & [\,] \\ a' & \mapsto & cons(a', cons(a, l)) \end{array} \right. \\ pop : (l, a) \end{array} \right. \end{array} \right.\right|\!\!\right)$$

As can be seen the stack is implemented using a list which holds the state of the unfold.

When the unfold is destructed with the *push* destructor a function is generated. The function takes an argument and puts it onto the head of the list. Thus, whenever the *push* destructor is used, an argument must be supplied in addition to the unfold. Pushing a -1 empties the stack. The *pop* destructor returns the head of the list and changes the state to the tail of the list. If the list is empty then *pop* returns a -1 signalling the stack is empty.

A map for the datatype *stack* requires two functions for the bivariant type parameter A. One function has type $f : A \to B$, the other function has type $g : B \to A$. Function f maps a value retrieved from the stack by using the *pop* destructor. Function g maps a value placed on the stack using the *push* destructor. If g is not the inverse of f then the results will be interesting!

1.2 Defined functions

There is a mechanism for defining (naming) functions in **charity**. Once a function is defined its name can be used to invoke the function. For example the map function that appears above which maps booleans to the integers 0 and 1 can be named with the following syntax:

$$\text{def } \mathit{boolToInt} \;= L \mapsto \mathit{list}\left\{ \begin{array}{lcl} \mathit{false} & \mapsto & 0 \\ \mathit{true} & \mapsto & 1 \end{array} \right\} L$$

The function is named *boolToInt* and its input parameter is named L. To call the function on a list one would write:

$$\mathit{booToInt}\ [\mathit{false}, \mathit{true}]$$

A function definition can also include function variables which must be instantiated when the function is called. For example,

$$\text{def } compose\{f, g\} = x \mapsto f\, g\, x$$

defines function *compose* that takes two function variables, f and g. When *compose* is called the function variables must be instantiated, as the following example illustrates:

$$compose\{x \mapsto boolToInt\, x,\ y \mapsto [y]\}\ \mathit{false}$$

1.3 Overview of patterns

A simple pattern in **charity** is a variable, the unit or a constructor followed by a variable or unit pattern. **Charity** core term logic supports only simple patterns. The patterned term logic of **charity** supports a much wider variety of patterns. These are discussed in more detail in chapter 2. In this section, we give a broad overview of patterns and some of the different types of patterns.

Peyton-Jones suggests in (Peyton Jones & Wadler, 1987) that a pattern matching compiler should support certain features of patterns such as overlapping patterns, nested patterns, multiple arguments, repeated variables, and boolean guards. To this list we add generalized guards. These features are discussed in the following subsections.

1.3.1 Overlapping patterns

Two patterns overlap if there is a term which matches both patterns. In functional languages evaluation depends upon the syntactical ordering of the patterns; the patterns are checked from top to bottom and the first one that matches is used. An example of overlapping patterns is:

$$\left\{ \begin{array}{lll} \mathit{zero} & \mapsto & x \\ \_ & \mapsto & \mathit{succ}\, x \end{array} \right\}$$

If the ordering of the cases was reversed in this example the *zero* case would never be evaluated because it would be completely overlapped by the "_" case. Generally the presence of a redundant pattern indicates an oversight on the part of the programmer. **Charity** allows overlapping patterns.

1.3.2 Nested patterns

Patterns are *nested* when one (or more) patterns are contained within another pattern. Simple patterns are not nested. As an example of nested patterns consider:

$$\mathit{cons}(\mathit{zero}, \mathit{cons}(\mathit{succ}\, \mathit{zero}, \mathit{nil}))$$

The constructor patterns *zero* and $\mathit{cons}(\mathit{succ}\, \mathit{zero}, \mathit{nil})$ are nested inside the first *cons*. The patterns *succ zero* and *nil* are nested inside the second *cons*. **Charity** supports this concept.

1.3.3 Multiple arguments

Multiple arguments in **charity** are grouped into pairs. The pair is considered a pattern and each element of the pair can also be a pattern. The case function below demonstrates this:

$$\left\{ \begin{array}{lll} (\mathit{succ}\, \mathit{zero}, \mathit{cons}(\mathit{zero}, \mathit{nil})) & \mapsto & \mathit{true} \\ (\mathit{succ}\, \mathit{zero}, \_) & \mapsto & \mathit{false} \\ \_ & \mapsto & \mathit{true} \end{array} \right\}$$

The idea of multiple arguments has been extended in **charity** to allow record patterns.

1.3.4 Repeated variables

The repeated use of a variable within a pattern implies that each use of the *repeated variable* is equivalent to each other use. As an example consider:

$$\left\{ \begin{array}{lcl} nil & : & () \mapsto nil \\ cons & : & \left| \begin{array}{lcl} ((x,x),L) & \mapsto & cons(x,L) \\ ((\_,\_),L) & \mapsto & L \end{array} \right. \end{array} \right\}$$

The pattern $((x,x),L)$ contains the variable x repeated once. The problem here is that **charity** needs to be able to test for equality on inputs of arbitrary types. In general, this is not possible since the types could be infinite; therefore, repeated variables are not allowed in **charity**.

1.3.5 Boolean guards

A programming language that has pattern matching with boolean guards allows the programmer to guard the application of a pattern with a conditional equation. This can lead to a non-exhaustive set of patterns since the guard may screen out a pattern that would otherwise match. As an example of the implementation of guards in **charity** the example given at the beginning of the chapter is modified below to apply function f if the first two elements of a list are equal. If the first two elements are not equal than failure is returned (*ff*).

$$\left\{ \begin{array}{lcl} (cons(a,\_), cons(b,\_)) \mid eq(a,b) & \mapsto & ss(f\ a) \mid .. \mapsto ff \\ \_ & \mapsto & ff \end{array} \right\} (L, L')$$

The **charity** implementation of boolean guards is described in detail in section 3.2.

1.3.6 Generalized guards

There is the possibility that an argument can be well-typed yet fail to match any of a function's patterns. For example, $(\mathit{zero}, \mathit{cons}(\mathit{false}, \mathit{nil}))$ does not match any of the phrases in the function below:

$$\left\{\begin{array}{lcl} (\mathit{zero}, \mathit{nil}) & \mapsto & \mathit{cons}(\mathit{false}, \mathit{nil}) \\ (\mathit{succ}\, x, \mathit{cons}(y, L)) & \mapsto & L \end{array}\right\}$$

This next case function, however, matches any well-typed input:

$$\left\{\begin{array}{lcl} (\mathit{zero}, \mathit{nil}) & \mapsto & \mathit{cons}(\mathit{false}, \mathit{nil}) \\ (\mathit{succ}\, x, \mathit{cons}(y, L)) & \mapsto & L \\ (\mathit{zero}, \mathit{cons}(x, L)) & \mapsto & \mathit{cons}(\mathit{false}, \mathit{cons}(\mathit{true}, \mathit{nil})) \\ (\mathit{succ}\, \_, \mathit{nil}) & \mapsto & \mathit{cons}(\mathit{true}, \mathit{cons}(\mathit{false}, \mathit{nil})) \end{array}\right\}$$

We say this is a complete set of pattern phrases. In **charity** not all sets of pattern phrases need be complete. However, all possible inputs to a function must, at some point, match a pattern. An incomplete set of pattern phrases can be used to guard a pattern, much like the conditional guards described above; this allows the programmer to generalize guards to functions of any type. By way of example we will guard the second pattern phrase of the example above with a function containing an incomplete set of pattern phrases:

$$\left\{\begin{array}{lcl} (\mathit{zero}, \mathit{nil}) & \mapsto & \mathit{cons}(\mathit{false}, \mathit{nil}) \\ (\mathit{succ}\, x, \mathit{cons}(y, L)) & \mapsto & \{\mathit{ff} \mapsto L\} f(x, y) \\ (\mathit{zero}, \mathit{cons}(x, L)) & \mapsto & \mathit{cons}(\mathit{false}, \mathit{cons}(\mathit{true}, \mathit{nil})) \\ (\mathit{succ}\, \_, \mathit{nil}) & \mapsto & \mathit{cons}(\mathit{true}, \mathit{cons}(\mathit{false}, \mathit{nil})) \end{array}\right\}$$

where $f(\mathit{zero}, 0) = \mathit{ss}(0)$. The pattern phrase set

$$\mathit{ff} \mapsto L$$

in the example above is incomplete. The term

$$(succ\, zero, cons(0, nil))$$

initially matches pattern

$$(succ\, x, cons(y, L))$$

but $f(zero, 0)$ evaluates to $ss(0)$ which does not match the inner pattern phrase. This causes the match on the outer pattern to fail also. This implies the above set of pattern phrases is incomplete since an input exists that does not match any of the patterns. Another pattern phrase must be added to make the set complete:

$$\left\{ \begin{array}{lcl} (zero, nil) & \mapsto & cons(false, nil) \\ (succ\, x, cons(y, L)) & \mapsto & \{ff \mapsto L\} f(x, y) \\ (zero, cons(x, L)) & \mapsto & cons(false, cons(true, nil)) \\ (succ\, \_, nil) & \mapsto & cons(true, cons(false, nil)) \\ \_ & \mapsto & nil \end{array} \right\}$$

Now, the input $(succ\, zero, cons(0, nil))$ will match the last pattern.

No other major functional programming language offers this generalized guard facility. The concept of complete sets of pattern phrases that underlies this facility is defined in chapter 4 along with more substantial examples of the use of generalized guards.

1.3.7 Matching patterns

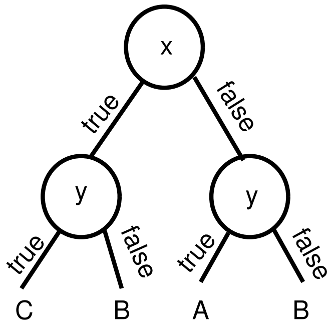
An argument that is tested for a match with a pattern is called the discriminant. A pattern is always associated with some code to be executed if the discriminant matches the pattern. This code is referred to as the righthand side (rhs). The pattern-rhs pair is called a pattern phrase. A set of pattern phrases is a group of pattern phrases arranged in a priority ordering. In this thesis, the priority ordering is always syntactic with priority decreasing from top to bottom.

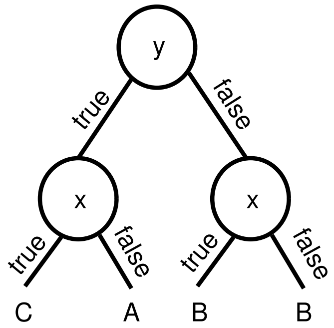
A set of pattern phrases reflects a parallel semantics. One can imagine the parallel semantics being implemented on a set of machines with each machine assigned to a constructor in a pattern. Given arguments or *discriminants* to match with the patterns, each machine evaluates the appropriate argument. If the constructor matches then a match success is indicated, otherwise a match fail is indicated and the pattern will not match. The rhs of a pattern phrase in which all the constructors of the pattern are matched is a candidate for selection. The rhs that is finally selected is the one which has a pattern matching the arguments and which comes first in the priority ordering (usually given by the syntactic ordering of the patterns). However, in order to know that no higher priority pattern matched all matches on these patterns must have failed. In the case of evaluations which are nonterminating, this information will not be forthcoming. This notion of the parallel semantics is formalized in the papers (Laville, 1987) and (Simpson & Cockett, 1992).

1.3.8 Translating patterns

The translation process sequentializes the above described parallel semantics. The sequentialization of patterns can be thought of as constructing a decision tree, with discriminants as the nodes, constructors as the branches, and rhs's as the leaves. For non-trivial sets of pattern phrases, there is more than one possible sequentialization. For example, the following set of pattern phrases can be represented by two different decision trees:

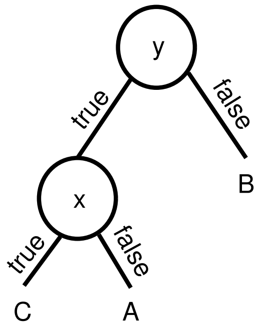
$$\left\{ \begin{array}{lcl} (\mathit{false}, \mathit{true}) & \mapsto & A \\ (\_, \mathit{false}) & \mapsto & B \\ (\_, \_) & \mapsto & C \end{array} \right\}$$





The decision tree on the left evaluates the x argument first; the tree on the right evaluates the y argument first.

Laziness can be introduced by reducing the number of evaluations required to ascertain the correct rhs to return. The following tree is lazier than the above two trees:



In general, because of the possibility of non-termination, it is only possible to preserve the parallel semantics in a particular class of patterns called *sequentializable* patterns. With sequentializable patterns, it is possible to find an order to sequentially evaluate arguments such that a value is returned if and only if a value would have been returned using the parallel semantics. In a sequentialized version we need to decide which subpattern to check first. When evaluating the following non-sequentializable example, if we happen to choose a nonterminating argument to evaluate first then no value will be returned. However, with the parallel semantics, if the other argument had evaluated to *true* a value would have been returned.

$$\begin{array}{llcl} \mathit{false} & \mathit{false} & \mapsto & \mathit{false} \\ \_ & \_ & \mapsto & \mathit{true} \end{array}$$

Given that it is not possible to always preserve the parallel semantics of patterns after sequentialization, some authors would like to select an evaluation order of arguments (usually left to right) and then have the translation and any optimization maintain the termination properties of this evaluation order (Augustsson, 1985; Peyton Jones & Wadler, 1987; Sekar, Ramesh, & Ramakrishnan, 1992). This is done to facilitate program proofs and to provide a guide for the programmer. The problem of preserving parallel semantics has been solved in **charity** since **charity** is strongly normalizing and termination always occurs.

Augustsson (Augustsson, 1985) is cited in several sources (Peyton Jones & Wadler, 1987) (Sekar et al., 1992) (Simpson & Cockett, 1992) as having the earliest published translation scheme for pattern phrases. Obviously it was not the first existing scheme since pattern matching was incorporated in *Hope* (1980) and before that in *SASL* (1976). Augustsson was concerned with implementing pattern matching in a specific language, *LML* (lazy meta language). His patterns were very basic; a pattern was either a variable or a constructor pattern. He allowed overlapping patterns but did not allow higher priority patterns to completely overlap lower priority patterns, using syntactic order as the basis of priority. Interestingly, though, he did not insist on exhaustive patterns. Thus every group of pattern phrases had an implicit default case at the bottom. If this default case was ever reached during execution it caused a runtime error.

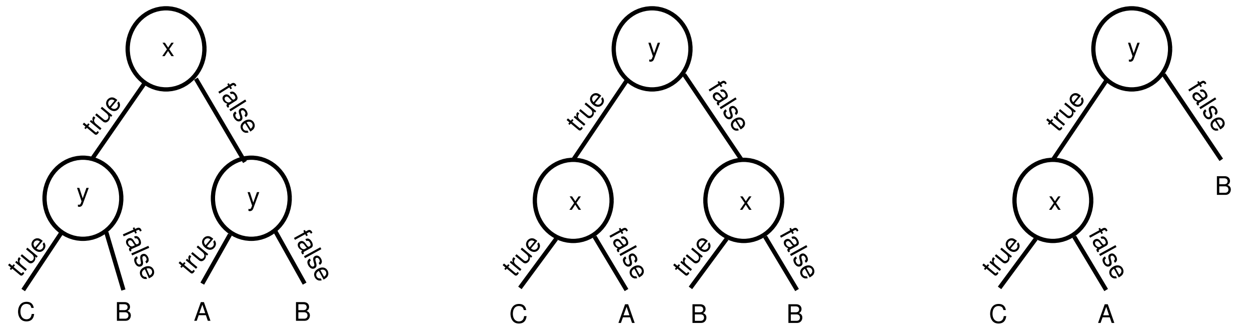
Wadler (Peyton Jones & Wadler, 1987) gives an algorithm very similar to Augustsson's for translating pattern phrases. He developed a general procedure for implementing pattern matching in functional languages. His set of patterns is richer than Augustsson's; product patterns and sum patterns are allowed.

It is not surprising that neither Augustsson's nor Wadler's algorithm yield an optimally lazy tree. In the past few years there have been papers whose goal has been to optimize laziness in decision trees.

One of these is a paper by Sekar et al. (Sekar et al., 1992). The authors propose a translation scheme based on the *indices* of a pattern. Intuitively an index is an argument that must be examined to determine a pattern match. The translation proceeds by choosing an index to examine next and then setting up a case statement to select among the constructors at that index. The translation proceeds recursively under each constructor using only those patterns that could be reached by matching with the constructor.

Using this method a pattern phrase that begins with a variable can become part of several different subtranslations. This is because a variable will match any constructor. Thus, when the translation is complete, there may be several different routes to the same rhs. However, this method does not repeat decisions as Wadler's and Augustsson's methods do.

What Sekar et al. noticed is that, for a node in a decision tree whose immediate subnodes are all identical, it is possible to "pull up" the subnodes without affecting the termination properties of the original tree. This always results in a tree that is never any larger than the original tree and is often smaller. An example is given below. The y nodes are pulled up in the first tree which gives the second tree. In this tree the right side always returns B so the decision on x can be eliminated. This results in the third tree which is smaller than the original tree.



Simpson and Cockett present a translation algorithm that is similar to Sekar's method but it does not rely on indices; it simply proceeds in a left to right fashion. Once the pattern phrases have been sequentialized the authors use a decision tree theory to reintroduce as much laziness as possible (thus removing some of the extraneous rhs's). The decision tree theory encompasses Sekar's "pull up" method. In addition, it shows how trees can be transposed in order to be able to apply the "pull up" method more often. They also show that trees with repeat decisions in a path from root to node can be simplified. Their claim is that the translation algorithm, coupled with optimization, is fully lazy for sequentializable definitions and also improves upon the termination properties for nonsequentializable definitions.

Chapter 2

The Patterns of Charity

In **charity** core term logic, which is the term logic used for translation into combinators, the only permissible patterns are variables, the unit and simple constructor patterns. To improve the productivity of the programmer and the readability of **charity** programs more complex patterns are provided at the user interface. Section 2.1 introduces rules for constructing more complex patterns: recursively built constructor and record patterns along with range patterns for the builtin datatypes of integers and characters. The usefulness of these patterns is further enhanced by a set of extensions to the basic set of patterns given in section 2.2. Rules for matching input values with patterns are given in section 2.3.

Throughout this chapter and the remainder of the thesis, the generic inductive and coinductive datatypes, L and R respectively, will be used to represent arbitrary datatypes. They are given below.

- L has n constructors with types:

$$\text{data } L(A) \to C = \left| \begin{array}{lcl} c_1 & : & E_1(A, L(A)) \to L(A) \\ & \vdots & \\ c_n & : & E_n(A, L(A)) \to L(A) \end{array} \right.$$

- R has r first order destructors and $n-r$ higher order destructors with types:

$$\text{data } C \rightarrow R(A) = \left| \begin{array}{lcl} d_1 & : & R(A) \rightarrow F_1(A, R(A)) \\ & \vdots & \\ d_r & : & R(A) \rightarrow F_r(A, R(A)) \\ d_{r+1} & : & E_{r+1}(A) \times R(A) \rightarrow F_{r+1}(A, R(A)) \\ & \vdots & \\ d_n & : & E_n(A) \times R(A) \rightarrow F_n(A, R(A)) \end{array} \right.$$

The symbol A, in both the above, is an r-tuple of type variables where $A_1, \ldots, A_p$ are covariant, $A_{p+1}, \ldots, A_q$ are contravariant, and $A_{q+1}, \ldots, A_r$ are bivariant.

We will also assume we have available previously defined symbols in a "symbol table" containing discrete sets of symbols. The classification of a symbol will be given by set membership. For instance,

$$L \in \text{types}$$

indicates that L is a datatype. For symbols that are typed, such as those representing constructors and destructors, the set membership notation will be expanded to include the type. For example the following two statements

$$\begin{array}{c} \mathit{false} \in \text{cstr}(\mathit{bool}) \\ \mathit{head} \in \text{dstr}(\mathit{stream}) \end{array}$$

indicate that *false* is a constructor of type *bool* and *head* is a destructor of type *stream*. The type indicators may be left off when they are not needed. For example,

$$\begin{array}{c} \mathit{false} \in \text{cstr} \\ \mathit{head} \in \text{dstr} \end{array}$$

Function symbols are also described using set membership. For instance

$$f \in \text{comb}$$

indicates that f is a function symbol. The description

$$f \in \text{comb}([], S \rightarrow T)$$

indicates a function,f, that has no function variables, accepts input of type S and generates output of type T. For functions with function variables, their types will be shown inside the [ ]s. For example,

$$g \in \text{comb}([S_0 \rightarrow T_0, S_1 \rightarrow T_1], S \rightarrow T)$$

The function g takes two functions as arguments.

Finally, we will need an infinite supply of variables for each type. A variable of type T is indicated by

$$v \in \text{var}(T)$$

Function variables can be indicated similarly:

$$f \in \text{var}(S \rightarrow T)$$

We may abbreviate this syntax by omitting the type information when it is obvious.

2.1 Basic charity patterns

Figure 2.1 describes how patterns are built in the patterned term logic. The base patterns are the variable, function variable and unit patterns. The constructor, pair and record patterns evolve from these base patterns.

variable	$$\frac{v \in \mathrm{var}(T) \quad v \text{ is new}}{v : T}$$
function variable	$$\frac{f \in \mathrm{var}(S \to T) \quad f \text{ is new}}{f : S \to T}$$
unit	$$\frac{}{() : 1}$$
constr	$$\frac{p : E_i(A, L(A)) \quad c_i \in \mathrm{cstr}(L) \quad 1 \leq i \leq n}{c_i\, p : L(A)}$$
pair	$$\frac{p : T \;,\; p' : T'}{(p_1, p_2) : T \times T'}$$
record	$$\frac{\begin{array}{ll} & p_j \in \mathrm{var} \\ p_i : F_i(A, R(A)) & i = 1..r \\ p_j : E_j(A) \to F_j(A, R(A)) & j = r+1..n \end{array}}{(d_i : p_i, d_j : p_j) : R(A)}$$
int range	$$\frac{i, j \in \mathbb{Z} \cup \{-\infty, \infty\} \quad i \leq j}{i..j : int}$$
char range	$$\frac{c_1, c_2 \in \mathrm{ascii} \quad c_1 \leq c_2}{c_1..c_2 : char}$$

Figure 2.1: **Charity** patterns

The rules are used to construct a proof that a particular syntactic entity is a valid **charity** pattern. To prove that $() : 1$ is a valid pattern we need merely to invoke the *unit* rule:

$$\frac{}{() : 1}\ \textit{unit}$$

Variable patterns can be proved with similar ease:

$$\frac{x \in \mathrm{var}(T) \quad x \text{ is new}}{x : T}\ \textit{variable}$$

Function variable pattern proofs are identical to variable pattern proofs with the exception of the typing:

$$\frac{g \in \mathrm{var}(S \to T) \quad g \text{ is new}}{g : S \to T}\ \textit{variable}$$

Note that, because of their type, function variables can only be used in the construction of higher order record patterns.

The *new* condition placed on variable and function variable patterns ensures that any variable introduced into a pattern will be unique. This prevents the occurrence of repeated variables in a pattern (see section 1.3). In the patterned term logic type theory, see section 3.1.1, the *new* condition will be relaxed somewhat so that variables with the same name can be used as long as they are not in the same scope.

A constructor pattern is created by prefixing another pattern, of the correct type, with a constructor name. A simple constructor pattern uses a base pattern. The proof:

$$\dfrac{\dfrac{}{():1}\,unit \qquad nil : 1 \to L(A) \quad nil \in \mathrm{cstr}(list)}{nil() : L(A)}\,constr$$

establishes that $nil()$ is indeed a simple constructor pattern.

Pair patterns can be used in conjunction with constructors to create complex constructor patterns. We have already shown that $x : T$ and $nil() : L(A)$ are patterns, therefore it follows from

$$\dfrac{x : T \quad nil() : L(T)}{(x, nil()) : T \times L(T)}\,pair$$

that their pairing is a pattern. Now, using the *cons* constructor, we get the proof:

$$\dfrac{(x, nil()) : T \times L(T) \quad cons \in \mathrm{cstr}(list) \quad cons : (T \times L(T)) \to L(T)}{cons(x, nil()) : L(T)}\,constr$$

that establishes $cons(x, nil())$ as a nested pattern (see section 1.3).

To create a record pattern of type $R(A)$ we need a pattern for every destructor of datatype R. The first order destructors may take any pattern except function variable patterns while higher order destructors are limited to function variable patterns. This is because functions cannot be further decomposed in an attempt to get a pattern to match. By limiting these patterns to a variable they will match with any function of the correct type.

Recall the declaration of the *stack* datatype given in chapter 1. From this a higher order record pattern can be developed as follows:

$$
\cfrac{\cfrac{f \in \mathrm{var}(SF(T) \to stack(T))}{f : SF(T) \to stack(T)}\;\textit{fun var} \qquad \cfrac{\cfrac{\cfrac{x \in \mathrm{var}(T)}{x : T}\;\textit{var}}{ss\,x : SF(T)}\;\textit{cstr} \qquad \cfrac{y \in \mathrm{var}(stack(T))}{y : stack(T)}\;\textit{var}}{(ss\,x, y) : SF(T) \times stack(T)}\;\textit{pair}}{(push : f, pop : (ss\,x, y)) : stack(T)}\;\textit{record}
$$

Note how the typing of x forces the typing of y and the function variable f.

The pair pattern, (p, p'), is an alternate form for the record pattern $(p_0 : p, p_1 : p')$. Pattern matching on pairs is thus very similar to pattern matching on records. This enables the programmer to pattern match on values of coinductive types without having to explicitly destruct them.

Range patterns allow the programmer to match more than one constructor with a single pattern. Ranges can only be used with constructors that are ordered. Currently the only two datatypes that can use range patterns are integers and characters. Integers are ordered in the usual manner while characters are ordered by their ascii value.

2.2 Extended charity patterns

A set of extended patterns (see figure 2.2) have been implemented in **charity**. These include the *don't care* variable; abbreviated constructor and record patterns; and a shorthand notation for constructors of the *list* datatype, range and string patterns. These serve to enhance the readability of **charity** code and the productivity of the programmer.

The *don't care* pattern is a variable pattern whose value will not be used (so no value need be transmitted). The don't care pattern cannot be used as a term; it

don't care	$v \equiv \_ \quad v \in \text{var} \quad v \text{ is new}$
constructor	$c() \equiv c \quad c \in \text{cstr}(L) \quad c : 1 \to L(A)$
record	$(d_1{:}p_1, \ldots, d_k{:}\_ , \ldots, d_n{:}v_n) \equiv (d_1{:}p_1, \ldots, d_n{:}v_n) \quad d_k \in \text{dstr}(R)$
list	$\mathit{cons}(p_1, \ldots, \mathit{cons}(p_n, \mathit{nil}) \ldots) \equiv [p_1, \ldots, p_n] \quad \mathit{nil} \equiv [\,]$
int range	$i..i \equiv i \quad -\infty..i \equiv ..i \quad i..\infty \equiv i.. \quad i \in \mathbb{Z}$
char range	$c..c \equiv c \quad \backslash x0..c \equiv ..c \quad c..\backslash x8f \equiv c.. \quad c \in \text{ascii}$
string	$[\backslash cc_1, \ldots, \backslash cc_n] : \mathit{list}(\mathit{char}) \equiv \text{“}c_1 \ldots c_n\text{”}$

Figure 2.2: Extended Patterns

serves merely as a placeholder. Use of the don't care pattern relieves the programmer of the burden of having to create variables that will not be used and improves the readability of programs.

The abbreviation which allows one to omit empty parentheses allows *nil*() to be written *nil*, and *true*() to be written as *true*.

The extended record pattern gives the programmer the option of not explicitly naming all destructors in a record pattern. Unnamed destructors are assumed to have a don't care pattern. For example, the following record pattern uses a don't care pattern as the pattern to the inner *tail* destructor.

$$(\mathit{head} : [..10, 8..], \mathit{tail} : (\mathit{head} : [\,], \mathit{tail} : \_)) : \mathit{stream}(\mathit{int})$$

Note that, in a record pattern for a recursive datatype, eventually a variable or don't care pattern must be employed to terminate the nesting of patterns. Since the value of the pattern is not required we can eliminate it along with its associated destructor:

$$(\mathit{head} : [..10, 8..], \mathit{tail} : (\mathit{head} : [\,])) : \mathit{stream}(\mathit{int})$$

Taking this idea to its logical conclusion one would expect to be able to replace the record pattern

$$(\mathit{head} : \_) : \mathit{stream}(A)$$

with the extended record pattern

$$() : \mathit{stream}(A)$$

However at the syntactic level, that is, without the typing information, this can be confused with the unit pattern and so is not allowed. In this case the record pattern can be replaced, in its entirety, with a don't care pattern.

The extended range patterns let the programmer avoid specifying the lower or upper limits of the range. The missing limit then defaults to the minimum or maximum value respectively. This is especially useful for integer ranges since it is impossible to enter ∞ on the keyboard!

The extended patterns are characterized by their ability to transform into one of the basic patterns without reference to any other pattern. This characteristic allows us to focus only on the basic patterns when discussing the patterned term logic type theory (chapter 3), the completeness of pattern abstractions (chapter 4), and the translation of patterned term logic into core term logic (chapter 5).

2.3 Matching patterns

Branching of program execution is controlled by the process of pattern matching. In this process, a discriminant (see section 1.3.7) is evaluated and compared to a pattern in a pattern phrase. If the discriminant matches the pattern then the term of the pattern phrase is executed. Pattern matching also provides access to nested values of the discriminant through variable substitution.

The rules for pattern matching are given in figure 2.3. The syntax

$$r \triangleright p \parallel \sigma$$

should be read as "discriminant r matches pattern p with substitution σ". A discriminant must have the same type as a pattern in order for matching to succeed.

Pattern matching on the variable, function variable, and unit patterns is straightforward. If the discriminant has the same type as the pattern then matching will succeed. Given a match of a discriminant on a variable or function variable the discriminant is substituted for the variable in the term guarded by the pattern. The mechanisms for making the substitutions into **charity** terms are given in section 5.1.1.

Consider a datatype for an n-ary tree:

$$\text{data } \mathit{tree}(A) \to C = \begin{array}{lcl} \mathit{leaf} & : & 1 \to C \\ \mathit{node} & : & A \times \mathit{list}(C) \to C \end{array}$$

Now, given the discriminant

$$\mathit{node}(\mathit{false}, \mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))) : \mathit{tree}(\mathit{bool})$$

and the pattern

$$\mathit{node}(x, \_) : \mathit{tree}(\mathit{bool}), \quad x : \mathit{bool}$$

variable	$$\frac{v : T \quad v \in \mathrm{var}(T) \quad r :_c T}{r \triangleright v \parallel t/v}$$
function variable	$$\frac{f : S \to T \quad f \in \mathrm{var} \quad g :_c S \to T}{g \triangleright f \parallel g/f}$$
unit	$$\frac{() : 1 \quad r :_c 1}{r \triangleright () \parallel}$$
constr	$$\frac{r :_c E_i(A, L(A)) \triangleright p \parallel \sigma \quad c_i \in \mathrm{cstr}(L) \quad 1 \leq i \leq n}{c_i\, r \triangleright c_i\, p \parallel \sigma}$$
pair	$$\frac{r_1 \triangleright p_1 \parallel \sigma_1 \quad r_2 \triangleright p_2 \parallel \sigma_2}{(r_1, r_2) \triangleright (p_1, p_2) \parallel \sigma_1 \cup \sigma_2}$$
record	$$\frac{\begin{array}{ll} r_i :_c F_i(A, R(A)) \triangleright p_i \parallel \sigma_i & \begin{array}{l} i = 1..r \\ j = r+1..n \\ p_j \in \mathrm{var} \end{array} \\ g_j :_c E_j \to F_j(A, R(A)) \triangleright p_j \parallel \sigma_j & \end{array}}{(d_i : r_i, d_j : g_j) \triangleright (d_i : p_i, d_j : p_j) \parallel \bigcup_{i=1}^{r} \sigma_i \bigcup_{j=r+1}^{n} \sigma_j}$$
int	$$\frac{r :_c \mathit{int} \quad i..j : \mathit{int} \quad i \leq r \leq j}{r \triangleright i..j \parallel}$$
char	$$\frac{r :_c \mathrm{char} \quad c_1..c_2 : \mathrm{char} \quad c_1 \leq r \leq c_2}{r \triangleright c_1..c_2 \parallel}$$

Figure 2.3: Matching Patterns

we check for a match. By the *constructor* rule we see that, since they share the same constructor, they will match if

$$(\mathit{false}, \mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))) \triangleright (x, \_)$$

By the *pair* rule, the above matches if $\mathit{false} \triangleright x$ and $\mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil})) \triangleright \_$. Variable x and constructor *false* are of the same type so we have

$$\mathit{false} \triangleright x \parallel \mathit{false}/x$$

Before checking the other part of the pair we shall convert $\_$ to a variable:

$$\_ \equiv v \quad v \in var(\mathit{list}(\mathit{tree}(\mathit{bool}))) \quad v \text{ is new}$$

Since v and $\mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))$ are of the same type they match and we get the substitution

$$\mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))/v$$

As we know that v will never be used, we need never actually collect this substitution. Now, working forwards through the rules, we gather the substitutions and arrive at

$$\mathit{node}(\mathit{false}, \mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))) \triangleright \mathit{node}(x, v) \parallel \mathit{false}/x, \mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))/v$$

To illustrate a failure to match we test a similar pattern,

$$\mathit{node}(x, [\_]) \quad ,$$

against the discriminant from the previous example. Working backwards from the *constructor* rule as before eventually we check if $\mathit{cons}(\mathit{leaf}, \mathit{cons}(\mathit{leaf}, \mathit{nil}))$ matches $[\_]$. We have

$$[\_] \equiv cons(\_, nil())$$

from the extended pattern equivalencies. The constructors are the same, but upon checking the right hand side of the constructor pattern, we see that

$$nil \not\triangleright cons(leaf, nil()) \quad .$$

Terms need only be evaluated as far as is required to determine whether or not they match a particular pattern. **Charity** eagerly evaluates all terms except records. For our purposes in this section we can assume **charity** has evaluated record terms to a sufficient depth to determine a match. As we will see in chapter 5, the patterned term logic translation algorithm forces a sufficient evaluation to occur.

Recall the *stack* datatype given in chapter 1. We will test for a match between the discriminant

$$\left(push : \left| \begin{array}{lcl} ff & \mapsto & [\,] \\ ss\,a & \mapsto & cons(a, [\,]) \end{array} \right. , pop : (\ldots, ss\,2) \right)$$

and the pattern

$$(push : f, pop : (y, ss\,x))$$

The dots $(\ldots)$ in the discriminant stand for the as yet to be evaluated portion of the record term. By the *record* rule, we need to check that the inner patterns of the record match the inner patterns of the discriminant. Using the *function variable* rule we get

$$\begin{array}{lcl} ff & \mapsto & [\,] \\ ss\,a & \mapsto & cons(a, [\,]) \end{array} \triangleright f \parallel \begin{array}{lcl} ff & \mapsto & [\,] \\ ss\,a & \mapsto & cons(a, [\,]) \end{array} / f$$

for the push destructor. Checking the components of the *pop* pair pattern we get

$$ss\,2 \triangleright ss\,x \parallel 2/x$$

and

$$\ldots \triangleright y \parallel \ldots/y \quad .$$

Therefore,

$$\left(push\!:\left| \begin{array}{lcl} ff & \mapsto & [\,] \\ ss\,a & \mapsto & cons(a,[\,]) \end{array} \right. , pop\!:(\ldots, ss\,2) \right) \triangleright (push\!:f, pop\!:(y, ss\,x)) \parallel 2/x, \ldots/y$$

Chapter 3

Patterned Term Logic

This chapter introduces and exemplifies the formal definition of the syntax and typing of the patterned term logic which was first developed in (Vesely, 1997). In addition, extensions to this syntax are given along with their definitions in terms of the patterned term logic. The patterned term logic type theory is used to derive "proofs" that expressions are well-formed.

Only the most basic patterns (the unit, variables, simple constructor patterns) are available in **charity** core term logic. Variables must be introduced one at a time and branching of program execution is allowed only through case functions having simple constructor patterns that are complete for any possible well typed input; that is, there are no constructors missing.

The patterned term logic allows the use of recursively built constructor and record patterns which do not necessarily have to match all possible input, that is, they may be incomplete. In addition, there are special patterns for integers and characters, and a syntactic extension for boolean patterns (guards). These features combine to provide **charity** with a pattern matching functionality that compares favourably in sophistication to that of popular functional programming languages.

Section 3.1 formalizes the patterned term logic in a type theory. Section 3.2

introduces a boolean pattern syntactic extension—the guard syntax—and shows how it is integrated into the type theory of the patterned term logic.

3.1 Patterned term logic

To show how patterns are used in **charity** we show how patterns are integrated into the patterned term logic. We achieve this by introducing the patterned term logic as a type theory.

A judgement in the type theory has the following general form:

$$\Gamma \vdash \varphi$$

Here Γ, called the declaration context, is a (possibly empty) set of pattern declarations, and φ is a piece of **charity** syntax which can be constructed given Γ. The symbol φ can be one of three things,

1. a term, $t : T$;
2. a function, $f : S \to T$; or
3. a pattern abstraction, $g \mid\!: S \to T$.

For a piece of syntax to be legal it must be possible to construct a judgement for it using the rules of the patterned term logic type theory. The type theory originates with the rule

$$\frac{}{\vdash () :_c 1} \;.$$

From this one can build valid declaration contexts using the rules given in figure 3.1; these can then be used to construct terms using rules found in figures 3.2 and 3.4. These rules are discussed and exemplified in the next three subsections.

Intuitively a complete function or pattern abstraction can accommodate all possible well-typed inputs. In the patterned term logic type theory, completeness of a term, function or patterned abstraction is denoted by the subscript c on the type signature prefix, for example, $g \mid:_c S \rightarrow T$. Completeness of terms and functions is based, ultimately, on the completeness of pattern abstractions. Because the notion of completeness is nontrivial we delay giving the rules for generating judgements of complete pattern abstractions until the next chapter. In this chapter, we give only a rule generating judgements of incomplete pattern abstractions.

3.1.1 Patterns in the patterned term logic

The type theory for patterned term logic introduces patterns on the left of the judgement by modifying the declaration context (see figure 3.1). The process is the same as the one described in section 2.1 except the patterns are now built within the declaration context.

Although we introduce patterns earlier than the rules for constructing terms, in fact, in the type theory, terms must be constructed before the patterns are introduced. Patterns are not terms in spite of having a similar syntax and they can always be distinguished as syntactic components of a term.

The requirement that variables and function variables be new, described in section 2.1, is replaced in the type theory by the requirement that the variable or function variable, v, not be in Γ. This requirement is less stringent but it still prevents repeated variables in a pattern. However, it does allow a variable of the same name to be used in different scopes. We exemplify this by way of an example derivation. Although this derivation only deals with a term variable the idea is also applicable to function variables.

First, using the *unit* and *var* rules from figure 3.2 along with rules for building *variable*

$$\frac{\Gamma \vdash \varphi \quad v \in \mathrm{var}(T) \quad v \text{ is not in } \Gamma}{\Gamma, v : T \vdash \varphi}$$

function variable

$$\frac{\Gamma \vdash \varphi \quad f \in \mathrm{var}(S \to T) \quad f \text{ is not in } \Gamma}{\Gamma\,,\; f : S \to T \vdash \varphi}$$

unit

$$\frac{\Gamma \vdash \varphi}{\Gamma\,,\; () : 1 \vdash \varphi}$$

constr

$$\frac{\Gamma\,,\; p : E_i(A, L(A)) \vdash \varphi \quad c_i \in \mathrm{cstr}(L) \quad 1 \leq i \leq n}{\Gamma\,,\; c_i\, p : L(A) \vdash \varphi}$$

pair

$$\frac{\Gamma\,,\; p : T\,,\; p' : T' \vdash \varphi}{\Gamma\,,\; (p, p') : T \times T' \vdash \varphi}$$

record

$$\frac{\Gamma\,,\; p_i : F_i(A, R(A))\,,\; v_j : E_j(A) \to F_j(A, R(A)) \vdash \varphi \quad \begin{array}{l} v_j \in \mathrm{var} \\ i = 1..r \\ j = r+1..n \end{array}}{\Gamma\,,\; (d_i : p_i, d_j : v_j) : R(A) \vdash \varphi}$$

int range

$$\frac{\Gamma \vdash \varphi \quad i, j \in \mathbb{Z} \cup \{-\infty, \infty\} \quad i \leq j}{\Gamma\,,\; i..j : \mathit{int} \vdash \varphi}$$

char range

$$\frac{\Gamma \vdash \varphi \quad c_1, c_2 \in \mathrm{ascii} \quad c_1 \leq c_2}{\Gamma\,,\; c_1..c_2 : \mathit{char} \vdash \varphi}$$

pattern abstraction

$$\frac{|_{i=1}^{m} \Gamma\,,\; p_i : S \vdash t_i : T}{\Gamma \vdash |_{i=1}^{m}\, p_i \mapsto t_i \mathrel{|\!:} S \to T}$$

Figure 3.1: Type Theory: Patterns

$$\textit{variable} \qquad \frac{\Gamma\,,\, v : T \vdash \varphi \quad v \in \text{var}}{\Gamma\,,\, v : T \vdash v :_c T}$$

$$\textit{unit} \qquad \frac{}{\vdash () :_c 1}$$

$$\textit{pair} \qquad \frac{\Gamma \vdash t :_c T\,,\, t' :_c T'}{\Gamma \vdash (t, t') :_c T \times T'}$$

$$\textit{record} \qquad \frac{\Gamma \vdash \begin{array}{ll} t_i :_c F_i(A, R(A))\,, & i = 1..r \\ g_j \models_c E_j(A) \to F_j(A, R(A)) & j = r+1..n \end{array}}{\Gamma \vdash (d_i : t_i, d_j : g_j) :_c R(A)}$$

$$\textit{application} \qquad \frac{\Gamma \vdash t :_c S\,,\, f :_c S \to T}{\Gamma \vdash f\ t :_c T} \quad \frac{\Gamma \vdash t :_c S\,,\, f : S \to T}{\Gamma \vdash f\ t : T}$$

Figure 3.2: Type Theory: Terms

patterns, we generate a judgement with variable v in the context:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () :_c 1}\ \textit{unit term} \qquad v \in \textit{var}(A \times \textit{list}(A))}{v : A \times \textit{list}(A) \vdash ()}\ \textit{var patt}}{v \vdash v :_c A \times \textit{list}(A)}\ \textit{var term}}{\textit{cons}\, v : \textit{list}(A) \vdash v}\ \textit{cstr patt}$$

At this point we could not introduce another variable v into the declaration context because v is present in the constructor pattern $\textit{cons}\, v$. Continuing, we get

$$\frac{\textit{cons}\, v \vdash v : \textit{list}(A) \to A \times \textit{list}(A)}{\vdash \textit{cons}\, v \mapsto v : \textit{list}(A) \to A \times \textit{list}(A)}\ \textit{patt abs}$$

Now, we can introduce v back into the declaration context.

$$\frac{\vdash \mathit{cons}\, v \mapsto v \quad v \in \mathrm{var}(B)}{v : B \vdash \mathit{cons}\, v \mapsto v}\ \mathit{var\ patt}$$

The term variable, v, is in the scope of the pattern $\mathit{cons}\, v$ and its value will be determined by the value of the variable in $\mathit{cons}\, v$. In the above case, the outer pattern, v, will have no effect on the term variable.

Higher order destructors in record patterns must have function variables as their patterns. This is because functions cannot be further decomposed in an attempt to get a pattern match. By limiting the pattern to a variable there will be a match with any function of the correct type.

The *pattern abstraction* rule does not introduce patterns into the declaration context but rather it shifts patterns from the declaration context, Γ, to the syntactic side of the judgement, φ. Each pattern thus moved is associated with exactly one term by the $\mapsto$ arrow to create a pattern phrase. The *pattern abstraction* rule creates one or more pattern phrases per invocation. The set of pattern phrases is called a pattern abstraction. A pattern abstraction may be complete or incomplete: completeness is determined with the machinery given in chapter 4.

To illustrate the idea of the patterned term logic rules for building patterns and pattern abstractions we will derive the judgement below.

$$\vdash \left| \begin{array}{lll} (\mathit{false}(), x) & \mapsto & x \\ y & \mapsto & p_1\, y \end{array} \right. \models_c \mathit{bool} \times A \to A$$

The proof

$$
\dfrac{\dfrac{}{\vdash () :_c 1}\,\textit{unit term} \qquad x \in \mathrm{var}(A)}{\dfrac{\dfrac{\dfrac{\dfrac{x : A \vdash ()}{x \vdash x :_c A}\,\textit{var term}}{x\,,\,() : 1 \vdash x :_c A}\,\textit{unit patt}}{x\,,\,\textit{false}() : \textit{bool} \vdash x :_c A}\,\textit{cstr patt}}{(\textit{false}(), x) : \textit{bool}\times A \vdash x :_c A}\,\textit{pair patt}}\,\textit{var patt}
$$

delivers the prerequisite judgement for the first pattern phrase. The prerequisite judgement for the second pattern phrase is given by

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () :_c 1}\,\textit{unit term} \qquad y \in \mathrm{var}(\textit{bool}\times A)}{y : \textit{bool}\times A \vdash ()}\,\textit{var patt}}{y \vdash y :_c \textit{bool}\times A}\,\textit{var term} \qquad p_1 \in \mathrm{dstr}(\times)}{y \vdash p_1\, y :_c A}\,\textit{appl term}
$$

These two judgements combine to deliver the desired pattern abstraction.

$$
\dfrac{(\textit{false}(), x) : \textit{bool}\times A \vdash x :_c A \qquad y : \textit{bool}\times A \vdash p_1\, y :_c A}{\vdash \left| \begin{array}{lcl} (\textit{false}(), x) & \mapsto & x \\ y & \mapsto & p_1\, y \end{array} \right| : \textit{bool}\times A \to A}\,\textit{patt abs}
$$

Extended patterns are available in the patterned term logic using the equivalences given in figure 2.2. However, the *new* requirement attached to the variable in the *don't care* pattern equivalence in that figure must be replaced with the *not in* Γ requirement described above.

3.1.2 Terms in the patterned term logic

The type theory builds term judgements using the rules given in figure 3.2. Because the *unit* rule is the only rule in the patterned term logic type theory that does not require an antecedent, it is always the first rule used when constructing a proof.

Pair and record term proofs are constructed in a very similar fashion to pair and record pattern proofs, the difference being that the patterns are composed of subpatterns while the terms are composed of subterms.

A variable term must be bound to a variable pattern. This is why the *variable* rule has a side condition on the pattern in the declaration context. The restriction on repeated variables in patterns does not apply to terms where a variable may appear many times.

The *application* rule applies a function to a term. If the function and term are both complete then the resulting term is also complete. If only the term is complete then the result of the application rule is an incomplete term. Only complete terms can be translated into legal core term logic terms. This is the only rule that results in a judgement of an incomplete term. This is described further in the subsection describing functions.

Some trivial examples of building term judgements appeared in the previous subsection. More substantial examples are described in the subsection on functions.

3.1.3 Extended terms

A set of extended terms for **charity** are given in figure 3.3. The extended terms are the abbreviated constructor term and a shorthand notation for *list* constructors and strings. The extended terms have the same equivalencies as their extended pattern counterparts given in section 2.2. In a similar fashion to the extended patterns, the extended terms improve the readability of **charity** code and enhance the productivity

constructor	$c() \equiv c \;\; c \in \mathrm{cstr}(L) \;\; c : 1 \rightarrow L(A)$
list	$cons(p_1, \ldots, cons(p_n, nil) \ldots) \equiv [p_1, \ldots, p_n] \quad nil \equiv [\,]$
string	$[\backslash cc_1, \ldots, \backslash cc_n] : list(char) \equiv$ "$c_1 \ldots c_n$"

Figure 3.3: Extended Terms

of the programmer.

3.1.4 Functions in the patterned term logic

Functions are applied to terms to create other terms. The rules for creating function judgements appear in figure 3.4.

Like its term counterpart, the term variable, the function variable may appear more than once on the right hand side of a judgement. Constructors and destructors are created with datatype definitions and function names are created with function definitions as described in chapter 1. The case, fold, map and unfold functions are particular syntactic structures that perform a well defined task particular to the type of input they receive. They are described briefly in chapter 1 and more fully in (Schroeder, 1997).

There are two versions of the case function rule: the complete version and the incomplete version. Recall from the earlier subsection on terms that the *application* rule was the only rule able to accept incomplete functions. Since the case function is the only function that may be incomplete the only allowable incomplete term is the case term.

The fold function must have a function g for every constructor c of the input *function variable*

$$\frac{\Gamma\,,\ f : S \to T \vdash \varphi \quad f \in \mathrm{var}}{\Gamma\,,\ f \vdash f :_c S \to T}$$

structor or *defined function*

$$\frac{\Gamma \vdash \varphi \quad s : S \to T \in \mathrm{cstr} \cup \mathrm{dstr} \cup \mathrm{comb}([\,]\,,\ S \to T)}{\Gamma \vdash s :_c S \to T}$$

defined function

$$\frac{\Gamma \vdash |_{i=1}^{m}\ g_i \ |\!:_c S_i \to T_i \quad f \in \mathrm{comb}([S_i \to T_i]\,,\ S \to T)}{\Gamma \vdash f\{|_{i=1}^{m} g_i\} :_c S \to T}$$

case

$$\frac{\Gamma \vdash g \ |\!:_c S \to T}{\Gamma \vdash \{g\} :_c S \to T} \qquad \frac{\Gamma \vdash g \ |\!: S \to T}{\Gamma \vdash \{g\} : S \to T}$$

fold

$$\frac{\Gamma \vdash |_{i=1}^{n}\ g_i \ |\!:_c E_i(A,T) \to T \quad c_i \in \mathrm{cstr}(L)}{\Gamma \vdash \{\!|\,|_{i=1}^{n} c_i : g_i\,|\!\} :_c L(A) \to T}$$

map

$$\frac{\Gamma \vdash \begin{array}{ll} |_{i=1}^{p}\ g_i^{+} \ |\!:_c A_i \to B_i, & i = 1..p \\ |_{j=p+1}^{q}\ g_j^{-} \ |\!:_c B_j \to A_j, & j = p+1..q \\ |_{k=q+1}^{r}\ g_k^{+} \ |\!:_c A_k \to B_k, & k = q+1..r \\ |_{k=q+1}^{r}\ g_k^{-} \ |\!:_c B_k \to A_k & T \in \mathrm{types} \end{array}}{\Gamma \vdash T \left\{ \begin{array}{c} g_i^{+}, \\ g_j^{-}, \\ g_k^{+} \ \&\ g_k^{-} \end{array} \right\} :_c T(A) \to T(B)}$$

unfold

$$\frac{\left|_{k=1}^{m} \ \Gamma\,,\ p_k : S \vdash \begin{array}{l} |_{i=1}^{r}\ t_i^{k} : F_i(A, R(A))\,, \\ |_{j=r+1}^{n}\ g_j^{k} \ |\!: E_j(A) \to F_j(A, R(A)) \end{array}\right. \qquad |_{k=1}^{m}\ p_k \mapsto () \ |\!:_c S \to 1}{\Gamma \vdash \left(\left|_{k=1}^{m} \ p_k \mapsto \begin{array}{l} |_{i=1}^{r}\ d_i : t_i^{k} \\ |_{j=r+1}^{n}\ d_j : g_j^{k} \end{array}\right.\right) :_c S \to R(A)}$$

Figure 3.4: Type Theory: Functions

datatype. The constructors and the functions are paired into phrases by the syntax $c : g$. By way of example we will build a judgement proof for a fold function that takes a lazy list as input and delivers the last element of the list, if it exists, that satisfies function *pred*. Suppose we have derived the following judgements:

$$\Gamma \vdash \mathit{pred} :_c A \to \mathit{SF}(A)$$

$$\Gamma \vdash \ () \ \mapsto \ \mathit{ff} \ \mid :_c 1 \to \mathit{SF}(A)$$

$$\Gamma \vdash \left| \begin{array}{lcl} (\mathit{fst} : \mathit{head}, \mathit{snd} : \mathit{ff}) & \mapsto & \mathit{pred}\,\mathit{head} \\ (\mathit{snd} : \mathit{result}) & \mapsto & \mathit{result} \end{array} \right| :_c \mathit{Lprod}(A, \mathit{SF}(A)) \to \mathit{SF}(A)$$

The fold itself will have type $\mathit{Llist}(A) \to \mathit{SF}(A)$. We will be folding over the datatype *Llist* so there are two constructors to consider:

- $\mathit{Lnil} : 1 \to \mathit{Llist}(A)$
- $\mathit{Lcons} : \mathit{Lprod}(A, \mathit{Llist}(A)) \to \mathit{Llist}(A)$

In a fold function the type of the range of the fold is substituted for the type of the range of the constructors. This means the pattern abstraction for *Lnil* must have type $1 \to \mathit{SF}(A)$ and for *Lcons* the type must be $\mathit{Lprod}(A, \mathit{SF}(A)) \to \mathit{SF}(A)$. The pattern abstractions we are given have the desired type. Thus we are able to conclude,

$$\Gamma \vdash \left\{ \begin{array}{lcl} \mathit{Lnil} & : & () \ \mapsto \ \mathit{ff} \\ \mathit{Lcons} & : & \left| \begin{array}{lcl} (\mathit{fst} : \mathit{head}, \mathit{snd} : \mathit{ff}) & \mapsto & \mathit{pred}\,\mathit{head} \\ (\mathit{snd} : \mathit{result}) & \mapsto & \mathit{result} \end{array} \right. \end{array} \right\} :_c \mathit{Llist}(A) \to \mathit{SF}(A)$$

The map function requires a function for every type variable of T that has negative or positive variance. (A bivariant variable of T actually requires two functions. One function maps the variable as if it were covariant; the other function maps it as if it were contravariant.) The functions are associated with the type variables through the declaration order of the type variables in the datatype declaration. Recall the *stack* datatype given in chapter 1. We will modify it slightly by adding a type variable to be used, perhaps, to indicate the quality of each datum in the stack:

$$\text{data } S \rightarrow qstack(A,Q) = \begin{array}{lcl} qpop & : & S \rightarrow S \times (A \times Q) \\ qpush & : & S \rightarrow A \times S \end{array}$$

Type variable A is bivariant and type variable Q is covariant. Assume we have three judgements below already available.

1. $\Gamma\,,\, a \vdash AB \,\mathrel{\vdots}_c\, A \rightarrow B$

2. $\Gamma\,,\, b \vdash BA \,\mathrel{\vdots}_c\, B \rightarrow A$

3. $\Gamma\,,\, q \vdash QR \,\mathrel{\vdots}_c\, Q \rightarrow R$

With these in hand, we construct judgements for the necessary pattern abstractions.

$$\cfrac{\cfrac{\cfrac{\Gamma\,,\, a \vdash AB \,\mathrel{\vdots}_c\, A \rightarrow B}{\Gamma\,,\, a \vdash AB\,,\, a :_c A}\,var}{\Gamma\,,\, a \vdash AB\, a :_c B}\,appl}{\Gamma \vdash a \mapsto AB\, a \,\mathrel{\vdots}_c\, A \rightarrow B}\,patt\ abs$$

Similarly we get

$$\Gamma \vdash b \mapsto BA\, b \,\mathrel{\vdots}_c\, B \rightarrow A$$

and

$$\Gamma \vdash q \mapsto QR\, q \,\mathrel{\vdots}_c\, Q \rightarrow R \ \ .$$

Now we can give a valid judgement for a map function on *qstack*:

$$\Gamma \vdash qstack \left\{ \begin{array}{l} a \mapsto AB\, a \ \&\ b \mapsto BA\, b\,, \\ q \mapsto QR\, q \end{array} \right\} :_c qstack(A,Q) \rightarrow qstack(B,R)$$

The position of the abstractions in the map function is critical since it is the order in which they appear that correlates them to a type variable. In the data definition of *qstack*, A is the first and Q is the second type variable in the type variable tuple. Thus the first pattern abstraction in the map function is for type variable A in the covariant position; the second is for type variable A in the contravariant position, and the final pattern abstraction is for type variable Q which only appears in a covariant position.

The $|_{k=1}^{m} p_k \mapsto () \models_c S \to 1$ antecedent in the *unfold* rule ensures that the patterns of the unfold match any possible input to the unfold function. The g_j^k functions are not required to be complete because the outer patterns are complete. This is explained in chapter 4. To illustrate the unfold rule in action we prove the following judgement in its entirety:

$$\vdash x \mapsto \left(\!\left| \begin{array}{lcl} \mathit{nil} & \mapsto & \left| \begin{array}{lcl} \mathit{push} & : & a \mapsto [a] \\ \mathit{pop} & : & ([\,], x) \end{array} \right. \\ \\ \mathit{cons}(a, l) & \mapsto & \left| \begin{array}{lcl} \mathit{push} & : & a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \\ \mathit{pop} & : & (l, a) \end{array} \right. \end{array} \right|\!\right) [\,] \models_c A \to \mathit{stack}(A)$$

This is an implementation of a stack. First we prove the judgement for the pattern abstraction of the *nil* pattern:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () :_c 1}\,\mathit{unit} \qquad x, a \in \mathrm{var}(A)}{x : A, a : A \vdash ()}\,\mathit{var}}{x, a, \mathit{nil} : \mathit{list}(A) \vdash ()}\,\mathit{cstr},\ \mathit{patt}\ \equiv}{x, a, \mathit{nil} \vdash \mathit{nil}() : \mathit{list}(A)}\,\mathit{cstr},\ \mathit{appl}}{x, a, \mathit{nil} \vdash [\,] :_c \mathit{list}(A)}\,\mathit{term}\ \equiv}{x, a, \mathit{nil} \vdash ([\,], x) :_c \mathit{list}(A) \times A}\,\mathit{var},\ \mathit{pair}$$

With that in hand, we retain the declaration context and prove the judgement for the term of the *nil* pattern:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x, a, \mathit{nil} \vdash ([\,], x) :_c \mathit{list}(A) \times A}{x, a, \mathit{nil} \vdash \mathit{nil}() :_c \mathit{list}(A)}\;\mathit{unit,\ cstr,\ appl}}{x, a, \mathit{nil} \vdash (a, \mathit{nil}()) :_c A \times \mathit{list}(A)}\;\mathit{var,\ pair}}{x, a, \mathit{nil} \vdash \mathit{cons}(a, \mathit{nil}()) :_c \mathit{list}(A)}\;\mathit{cstr,\ appl}}{x, a, \mathit{nil} \vdash [a] :_c \mathit{list}(A)}\;\mathit{term} \equiv}{x, \mathit{nil} \vdash a \mapsto [a] \mid\!:_c A \to \mathit{list}(A)}\;\mathit{patt\ abs}$$

Next, we prove the judgement for the term of the *cons* pattern:

$$\dfrac{\dfrac{\dfrac{}{\vdash () :_c 1}\;\mathit{unit} \qquad x, a \in \mathrm{var}(A) \quad l \in \mathrm{var}(\mathit{list}(A))}{x : A, a : A, l : \mathit{list}(A) \vdash ()}\;\mathit{var}}{x, a, l \vdash (a, l) :_c \mathit{list}(A) \times A}\;\mathit{var,\ pair}$$

With that in hand, we retain the declaration context and prove the judgement for the pattern abstraction of the *cons* pattern:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : A, a : A, l : \mathit{list}(A) \vdash (a, l) :_c \mathit{list}(A) \times A \quad a' \in \mathrm{var}(A)}{x, a, l, a' : A \vdash (a, l)}\;\mathit{var}}{x, a, l, a' \vdash (a, l) :_c A \times \mathit{list}(A)}\;\mathit{var,\ pair}}{x, a, l, a' \vdash \mathit{cons}(a, l) :_c \mathit{list}(A)}\;\mathit{str,\ appl}}{x, a, l, a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) :_c \mathit{list}(A)}\;\mathit{var,\ pair,\ str,\ appl}}{x, a, l \vdash a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \mid\!:_c A \to \mathit{list}(A)}\;\mathit{patt\ abs}$$

It remains to construct the pattern we will be using in the unfold:

$$\frac{x : A, a : A, l : \mathit{list}(A) \vdash \begin{array}{l} a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \mathrel{|\!:_c} A \to \mathit{list}(A)\,, \\ (l, a) :_c \mathit{list}(A) \times A \end{array}}{x, \mathit{cons}(a, l) : \mathit{list}(A) \vdash \begin{array}{l} a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \mathrel{|\!:_c} A \to \mathit{list}(A)\,, \\ (l, a) :_c \mathit{list}(A) \times A \end{array}} \ \textit{pair, cstr}$$

We now have the following judgements in hand:

- $x, a, \mathit{nil} \vdash ([\,], x) :_c \mathit{list}(A) \times A$
- $x, \mathit{nil} \vdash a \mapsto [a] \mathrel{|\!:_c} A \to \mathit{list}(A)$
- $x, \mathit{cons}(a, l) : \mathit{list}(A) \vdash \begin{array}{l} a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \mathrel{|\!:_c} A \to \mathit{list}(A)\,, \\ (l, a) :_c \mathit{list}(A) \times A \end{array}$

The following pattern abstraction is indeed complete but the rules for determining this are not given until the next chapter.

$$\left|\begin{array}{lcl} \mathit{nil}() & \mapsto & () \\ \mathit{cons}(a, l) & \mapsto & () \end{array}\right. \mathrel{|\!:_c} \mathit{list}(A) \to 1$$

The above are sufficient to give us the following:

$$x \vdash \left(\begin{array}{lcl} \mathit{nil} & \mapsto & \left|\begin{array}{lcl} \mathit{push} & : & a \mapsto [a] \\ \mathit{pop} & : & ([\,], x) \end{array}\right. \\ \\ \mathit{cons}(a, l) & \mapsto & \left|\begin{array}{lcl} \mathit{push} & : & a' \mapsto \mathit{cons}(a', \mathit{cons}(a, l)) \\ \mathit{pop} & : & (l, a) \end{array}\right. \end{array}\right) :_c \mathit{list}(A) \to \mathit{stack}(A)$$

We omit the proof of the judgement

$$x : A \vdash [\,]$$

as it is easily derived. Now, we use the *function application* rule followed by the *pattern abstraction* rule to finish off:

$$\vdash x \mapsto \left(\begin{array}{ll} nil & \mapsto \left| \begin{array}{ll} push & : a \mapsto [a] \\ pop & : ([\,], x) \end{array} \right. \\ \\ cons(a,l) & \mapsto \left| \begin{array}{ll} push & : a' \mapsto cons(a', cons(a,l)) \\ pop & : (l, a) \end{array} \right. \end{array} \right) [\,] \mathrel{|\!\!:_c} A \to stack(A)$$

3.1.5 Programs in the patterned term logic

A program is a piece of patterned term logic syntax that can be compiled (indirectly) into correct machine code. It is then either evaluated immediately, an *executable program*, or stored for later use, a *defined program*.

The rules for delivering programs from the patterned term logic type theory appear in figure 3.5. Defined programs are named functions. The function name and its typing information are added to the set of combinators in the symbol table (comb). Once a program is defined it can used in accordance with the rules for functions given in figure 3.4. There are two forms of the *defined program* rule; one is used for defining functions that have function variables, the other for defining functions with none. To illustrate the defined program rule we will construct a named function for composing two functions. The functions will be passed in as function variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () :_c 1}\,unit \quad f \in \mathrm{var}(S \to T) \;\; g \in \mathrm{var}(R \to S) \;\; x \in \mathrm{var}(R)}{f : S \to T, g : R \to S, x : R \vdash () :_c 1}\,var}{f, g, x \vdash f\,g\,x :_c T}\,variable}{f, g \vdash x \mapsto f\,g\,x \mathrel{|\!\!:_c} R \to T}\,patt\ abs \qquad o \notin \mathrm{comb} \cup \mathrm{cstr} \cup \mathrm{dstr} \cup \mathrm{types}}{\mathrm{def}\ o\{f, g\} = x \mapsto f\,g\,x : [S \to T, R \to S], R \to T}\,def$$

The *pattern abstraction* step in the above proof shows delivery of a judgement for a complete pattern abstraction. While the pattern abstraction is indeed complete the machinery for demonstrating the completeness is not given until the next chapter.

defined, with macros	$$\frac{g_i : S_i \to T_i \vdash g \mid\!:_c S \to T \quad \begin{matrix} f \notin \text{comb} \cup \text{cstr} \cup \text{dstr} \cup \text{types} \\ i = 1..m \end{matrix}}{\text{def } f\{g_i\} = g : ([S_i \to T_i], S \to T)}$$
defined, no macros	$$\frac{\vdash g \mid\!:_c S \to T \quad f \notin \text{comb} \cup \text{cstr} \cup \text{dstr} \cup \text{types}}{\text{def } f = g : ([], S \to T)}$$
executable	$$\frac{\vdash t :_c T}{t :_c T}$$

Figure 3.5: Type Theory: Programs

An executable program is a judgement of a complete term with an empty declaration context. The empty declaration context indicates that all variables in the term are bound. This is a necessary precondition for a successful evaluation.

3.2 Boolean guards

A shorthand has been established in functional languages such as Miranda[1] (Holyer, 1991) for placing boolean conditions on the execution of terms associated with patterns. This syntactic sugar, called guards, has been also implemented in **charity**.

An example of the **charity** guard syntax is:

$$p \mid t_b \mapsto t \mid .. \mapsto t'$$

This is to be interpreted as "when the input matches pattern p then execute term t if the predicate t_b evaluates to *true*, otherwise execute term t'". In other words the predicate t_b is guarding the pattern p. It is a more succinct way of writing:

[1]Miranda is a trademark of Research Software Limited

positive, negative	$$\frac{\Gamma \vdash t_b : bool\,,\ t : T\,,\ u \mathrel{\dot{:}} T}{\Gamma \vdash t_b \mapsto t \mid u \mathrel{\dot{:}} T \quad \Gamma \vdash \tilde{}\,t_b \mapsto t \mid u \mathrel{\dot{:}} T}$$
unconditional	$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \,..\, \mapsto t \mathrel{\dot{:}} T}$$
guarded pattern abstraction	$$\frac{\Gamma\,,\ p : S \vdash u \mathrel{\dot{:}} T}{\Gamma \vdash p \mid u \mathrel{\dot{:}} S \to T}$$

Figure 3.6: Type Theory: Guard Term Extension

$$p \mapsto \left\{ \begin{array}{lcl} true & \mapsto & t \\ false & \mapsto & t' \end{array} \right\} t_b$$

The guard syntax has been extended in **charity** to allow multiple guards on a pattern, to allow a *false* case and to allow negation of guards. The patterned term logic type theory extension for guards is given in figure 3.6. The type theory treats guards as terms with guard terms identified by their type signature, for example, $t_g \mathrel{\dot{:}} T$.

The *guarded pattern abstraction* rule is used to place the guard term in its proper place after a pattern. The "|" is the typical symbol used to indicate that a pattern is guarded.

Guards can be used to implement an `if elseif else` construct as witnessed by the example below. Given

$$cond1,\ cond2\ \in \mathrm{comb}([\,], A \times B \to bool)$$

and

$$f \in \mathrm{comb}([\,], A \times B \to T)$$

we build the judgements

$$x, y \vdash cond1(x, y)$$

$$x, y \vdash cond2(x, y)$$

$$x, y \vdash f(x, y)$$

using rules of the patterned term logic type theory discussed in the previous section. The construction of the guard proceeds from the innermost term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x, y \vdash f()}{x, y \vdash \,..\, \mapsto f(x, y)\,,\, y}\;\textit{uncond, var} \qquad x, y \vdash cond2(x, y)}{x, y \vdash cond2(x, y) \mapsto y \mid \,..\, \mapsto f(x, y)\,,\, x}\;\textit{pos, var} \qquad x, y \vdash cond1(x, y)}{x, y \vdash cond1(x, y) \mapsto x \mid cond2(x, y) \mapsto y \mid \,..\, \mapsto f(x, y)}\;\textit{pos}}{(x, y) \vdash cond1(x, y) \mapsto x \mid cond2(x, y) \mapsto y \mid \,..\, \mapsto f(x, y)}\;\textit{pattern pairing}}{\vdash (x, y) \mid cond1(x, y) \mapsto x \mid cond2(x, y) \mapsto y \mid \,..\, \mapsto f(x, y)}\;\textit{guarded patt abs}$$

Guards are not the same as patterned term logic terms; this is suggested by the type signature of guards. To simplify the translation of the patterned term logic into the core term logic, guard terms and abstractions are translated into basic patterned term logic terms first. The translation is straightforward and is given in figure 3.7.

The translation is illustrated by translating the above guarded abstraction into a pattern abstraction. The translation of the guarded abstraction proceeds from the outermost term:

positive	$[\![t_b \mapsto t \mid u]\!] = \left\{\begin{array}{lll} true & \mapsto & t \\ false & \mapsto & [\![u]\!] \end{array}\right\} t_b$
negative	$[\![\tilde{\ } t_b \mapsto t \mid u]\!] = \left\{\begin{array}{lll} true & \mapsto & [\![u]\!] \\ false & \mapsto & t \end{array}\right\} t_b$
unconditional	$[\![.. \mapsto t]\!] = t$
guarded pattern abstraction	$[\![p \mid u]\!] = p \mapsto [\![u]\!]$

Figure 3.7: Guard Translation

$$
\begin{array}{ll}
 & [\![(x,y) \mid cond1(x,y) \mapsto x \mid cond2(x,y) \mapsto y \mid .. \mapsto f(x,y)]\!] \\
= & (x,y) \mapsto [\![cond1(x,y) \mapsto x \mid cond2(x,y) \mapsto y \mid .. \mapsto f(x,y)]\!] \\
= & (x,y) \mapsto \left\{\begin{array}{lll} \text{true} & \mapsto & x \\ \text{false} & \mapsto & [\![cond2(x,y) \mapsto y \mid .. \mapsto f(x,y)]\!] \end{array}\right\} cond1(x,y) \\
= & (x,y) \mapsto \left\{\begin{array}{lll} \text{true} & \mapsto & x \\ \text{false} & \mapsto & \left\{\begin{array}{lll} \text{true} & \mapsto & y \\ \text{false} & \mapsto & [\![.. \mapsto f(x,y)]\!] \end{array}\right\} cond2(x,y) \end{array}\right\} cond1(x,y) \\
= & (x,y) \mapsto \left\{\begin{array}{lll} \text{true} & \mapsto & x \\ \text{false} & \mapsto & \left\{\begin{array}{lll} \text{true} & \mapsto & y \\ \text{false} & \mapsto & f(x,y) \end{array}\right\} cond2(x,y) \end{array}\right\} cond1(x,y)
\end{array}
$$

Chapter 4

Completeness

In the previous chapter, a rule building a judgement of a possibly incomplete pattern abstraction was given. In this chapter, section 4.2 extends the patterned term logic type theory with rules for building complete pattern abstractions. In section 4.1, we extend the notion of matching, first given in section 2.3, to include incomplete pattern abstractions.

Completeness in **charity** is more of a syntactic notion than a semantic one. The following case term contains a pattern abstraction that is incomplete because there is no *false* pattern.

$$\{true \mapsto t\}\, f\, x$$

However, if $f\,x$ always evaluates to *true* then the *false* case will never be required and so, in a sense, the pattern abstraction would be complete. The completeness described here, a notion which a compiler must be able to determine easily, cannot rely on knowing how terms behave.

Thus, a pattern abstraction, $g \models S \to T$, is certainly incomplete if there exists a term, $t : S$, that does not match at least one of the patterns in the abstraction.

However, it is not the case that pattern abstractions are complete whenever there is no such term.

The pattern abstraction below is incomplete:

$$\left.\begin{array}{lcl} (x,(true,y)) & \mapsto & t_1 \\ (x,(y,false)) & \mapsto & t_2 \\ (true,x) & \mapsto & t_3 \end{array}\right| : bool \times (bool \times bool) \to T$$

The term $(false,(false,true)) :_c bool{\times}(bool{\times}bool)$ does not match any of its patterns. On the other hand, the following pattern abstraction is complete, if $t_1, t_2, t_3 :_c T$:

$$\left.\begin{array}{lcl} (false,(x,true)) & \mapsto & t_1 \\ (x,(y,false)) & \mapsto & t_2 \\ (true,x) & \mapsto & t_3 \end{array}\right| : bool \times (bool \times bool) \to T$$

If any of the terms t_1, t_2 or t_3 were incomplete then the above pattern abstraction would be incomplete even though the patterns can, by themselves, match any well-typed input. This is because the incompleteness of a term admits the possibility that a match can fail on one of its pattern abstractions. If this occurs then the match also fails on the outer pattern.

The completeness of the above pattern abstraction can be verified by deriving a "proof" of it using type theory rules given in chapter four and the rules that will defined and exemplified in this chapter. In general, these verifications are likely to be time consuming for all but the most simple examples.

One way to guarantee completeness would be to require that all pattern abstractions in a **charity** program be complete. However forcing the programmer to always write complete pattern abstractions can be cumbersome. Consider the following example. Given two datatype declarations,

$$\text{data } states(A_1, A_2) \to C = \left|\begin{array}{lcccc} s_1 & : & A_1 & \to & C \\ s_2 & : & A_2 & \to & C \end{array}\right.$$

and

$$\text{data } actions \to C = \; a_1 \mid a_2 \mid a_3 \;\; : \;\; 1 \to C \;\; ,$$

along with the following functions and term,

$$\begin{aligned}
&h :_c actions \to SF(T),\\
&f :_c S \to actions,\\
&g :_c S' \to actions,\\
&a :_c states(S, S'),
\end{aligned}$$

we can build the case term

$$\left\{\begin{array}{lll}
s_1\, x & \mapsto & \left\{\begin{array}{lll} a_1 & \mapsto & ff \\ a_2 & \mapsto & h\, a_2 \\ a_3 & \mapsto & ff \end{array}\right\} f\, x \\
s_2\, x & \mapsto & \left\{\begin{array}{lll} a_1 & \mapsto & h\, a_1 \\ a_2 & \mapsto & ff \\ a_3 & \mapsto & ff \end{array}\right\} g\, x \\
_ & \mapsto & ff
\end{array}\right\} a \;\; .$$

The above case term is complete since all its constituent pattern abstractions are complete.

Note, however, that the *ff* term is executed in five of seven possible locations. It would be more succinct and convenient if the programmer could write instead

$$\left\{\begin{array}{lll}
s_1\, x & \mapsto & \left\{\; a_2 \;\; \mapsto \;\; h\, a_2 \;\right\} f\, x \\
s_2\, x & \mapsto & \left\{\; a_1 \;\; \mapsto \;\; h\, a_1 \;\right\} g\, x \\
_ & \mapsto ff &
\end{array}\right\} a \;\; .$$

This can be understood as meaning

`If` $a \triangleright s_1$ `and` $f\,x \triangleright a_2$ `then execute` $h\,a_2$

`else if` $a \triangleright s_2$ `and` $g\,x \triangleright a_1$ `then execute` $h\,a_1$

`else execute` *ff*

The inner case terms of the above example are generalized guards; they are guarding the outer pattern. If a match on the outer pattern succeeds but the match on the inner pattern fails then it is as if the outer match also failed.

As an example, let a evaluate to s_1 *false* and f *false* evaluate to a_3. Then s_1 *false* matches the first outer pattern and x is instantiated to *false*. But, since f *false* $= a_3$ it fails to match a_2. Continuing to look for a match, we see that discriminant a does not match the second outer pattern, $s_2\,x$, but it does match _. Thus *ff* is executed.

For another example, let a evaluate to $s_2\,5$ and $g\,5$ evaluate to a_1. Discriminant $s_2\,5$ does not match $s_1\,x$ but does match $s_2\,x$. The x becomes 5, and $g\,5$ evaluates to a_1. This matches a_1 in the case function and $h\,a_1$ is executed.

4.1 Matching patterns

To accommodate incomplete pattern abstractions we need to extend the notion of what it means for a discriminant to match a pattern. Given a pattern abstraction

$$|_{i=1}^{m}\, p_i \mapsto t_i \,\vdots\, S \to T$$

and a discriminant, $r :_c S$, the machinery presented in section 2.3 is insufficient to determine a match. If the statement

$$r \triangleright p_i \parallel \sigma_i$$

complete	$\dfrac{p : S \mapsto t :_c T \quad r :_c S \quad r \triangleright p \parallel \sigma}{r \triangleright_c p \mapsto t[\sigma]}$
possibly incomplete	$\dfrac{p : S \mapsto t : T \quad r \triangleright p \parallel \sigma \quad \begin{array}{c} t = \left\{ \ \vert_{i=1}^{m} p_i \mapsto t_i \ \right\} r' : T \\ \exists i \cdot r'[\sigma] \triangleright_c p_i \mapsto t_i[\sigma \cup \sigma_i] \end{array}}{r \triangleright_c p \mapsto t[\sigma]}$

Figure 4.1: Matching Patterns

is true but $t_i : T$ is incomplete and a match is not found in t_i then ultimately the match will fail at the top level.

Rules for matching discriminants and pattern phrases are given in figure 4.1. The statement

$$r \triangleright_c p \mapsto t[\sigma]$$

is read as "discriminant r matches the pattern p with substitution σ into term t".

The effect of the *complete* rule is identical to the matching rules given earlier in section 2.3. The only check that needs to be made is whether the discriminant matches the pattern. Any substitutions are made directly into t.

Recall from the previous chapter that, given the rules of the patterned term logic type theory, the only term that could possibly be incomplete was the case term. That is, a case function applied to a term. The *possibly incomplete* rule, then, must have a case term as the right hand side of the pattern phrase. If the discriminant of this case term finds a match in the pattern abstraction of the case function then the outer match succeeds. The converse is also true; if the case term discriminant fails to find a match then the outer match fails. An example of this was given at the beginning of the chapter.

4.2 Complete pattern abstractions

Rules for constructing judgements of complete pattern abstractions in the patterned term logic type theory are given in figure 4.2 and are explained in this section.

The rules for constructing patterns in the type theory (see section 3.1.1) had only to concern themselves with the pattern in isolation. All the rules discussed in this section potentially operate on more than one pattern or pattern phrase at a time. For instance, suppose we have the following judgements of complete terms:

$$z : B, y : A, x : A \vdash (y, x)\,,\,(x, y)\,,\,(x, x) :_c A \times A \quad x, y, z \in \text{var}$$

We can create a complete pattern abstraction by introducing variables as patterns using the *variable* rule:

$$z : B, y : A \vdash \left| \begin{array}{ccl} x & \mapsto & (y, x) \\ x & \mapsto & (x, y) \\ x & \mapsto & (x, x) \end{array} \right. |\!:_c A \to A \times A \quad x, y, z \in \text{var}$$

The resulting pattern abstractions must be complete because a variable pattern will match any term.

We can sequentialize patterns with these rules. For instance, we can use the *variable* rule with the result of the last example to generate

$$z : B \vdash \left| \begin{array}{cccccl} y & \mapsto & (x & \mapsto & (y, x)) \\ y & \mapsto & (x & \mapsto & (x, y)) \\ y & \mapsto & (x & \mapsto & (x, x)) \end{array} \right. |\!:_c A \to (A \to A \times A) \quad x, y, z \in \text{var}$$

Parentheses are used to contain the patterns and terms on the right of the expression. Parentheses group the type signatures correspondingly. The purpose of sequentializing patterns is to be able to collapse them later into a pair or record pattern. All the complete pattern abstraction rules apply to terms with or without sequential patterns. The type of a term with sequential patterns is indicated by the subscript

$$
\textit{var} \qquad \frac{\Gamma,\ v : S \vdash |_{i=1}^{m}\ t_i :_c T_{\rightarrow} \quad v \in \mathrm{var} \quad v \notin \mathit{patt}(t_i)}{\Gamma \vdash |_{i=1}^{m}\ v \mapsto t_i \mathrel{|\!:}_c S \rightarrow T_{\rightarrow}}
$$

$$
\textit{unit} \qquad \frac{\Gamma \vdash |_{i=1}^{m}\ t_i :_c T_{\rightarrow}}{\Gamma \vdash |_{i=1}^{m}\ () \mapsto t_i \mathrel{|\!:}_c 1 \rightarrow T_{\rightarrow}}
$$

$$
\textit{constr} \qquad \frac{\Gamma \vdash |_{j=1}^{m_i}\ p_j^i \mapsto t_j^i \mathrel{|\!:}_c E_i(A, L(A)) \rightarrow T_{\rightarrow} \quad c_i \in \mathrm{cstr}(L) \quad i = 1..n}{\Gamma \vdash |_{j=1}^{m_i}\ c_i\, p_j^i \mapsto t_j^i \mathrel{|\!:}_c L(A) \rightarrow T_{\rightarrow}}
$$

$$
\textit{pair} \qquad \frac{\Gamma \vdash |_{i=1}^{m}\ p_i \mapsto (p_i' \mapsto t_i) \mathrel{|\!:}_c T_0 \rightarrow (T_1 \rightarrow T_{\rightarrow})}{\Gamma \vdash |_{i=1}^{m}\ (p_i, p_i') \mapsto t_i \mathrel{|\!:}_c T_0 \times T_1 \rightarrow T_{\rightarrow}}
$$

$$
\textit{record} \qquad \frac{\Gamma, v^j \vdash |_{k=1}^{m}\ p_k^i \mapsto t_k \mathrel{|\!:}_c F_i(A, R(A)) \rightarrow T_{\rightarrow} \quad \begin{array}{l} i = 1..r \quad j = r+1..n \\ v^j \in \mathrm{var}(E_j(A) \rightarrow F_j(A, R(A))) \\ v^j \notin \mathit{patt}(t_k) \end{array}}{\Gamma \vdash |_{k=1}^{m}\ (d_i : p_k^i, d_j : v_k^j) \mapsto t_k \mathrel{|\!:}_c R(A) \rightarrow T_{\rightarrow}}
$$

$$
\textit{int} \qquad \frac{\Gamma \vdash |_{k=1}^{m}\ i_k..j_k \mapsto t_k \mathrel{|\!:} \mathit{int} \rightarrow T_{\rightarrow} \quad \forall n \in \mathbb{Z} \quad \exists k \cdot i_k \leq n \leq j_k}{\Gamma \vdash |_{k=1}^{m}\ i_k..j_k \mapsto t_k \mathrel{|\!:}_c \mathit{int} \rightarrow T_{\rightarrow}}
$$

$$
\textit{char} \qquad \frac{\Gamma \vdash |_{k=1}^{m}\ c_k..d_k \mapsto t_k \mathrel{|\!:} \mathrm{char} \rightarrow T_{\rightarrow} \quad \forall c \in \mathrm{ascii} \quad \exists k \cdot c_k \leq c \leq d_k}{\Gamma \vdash c_k..d_k \mapsto t_k \mathrel{|\!:}_c \mathrm{char} \rightarrow T_{\rightarrow}}
$$

$$
\textit{weakening} \qquad \frac{\Gamma, p : S \vdash t : T_{\rightarrow} \quad \Gamma \vdash |_{i=1}^{m}\ p_i \mapsto t_i \mathrel{|\!:}_c S \rightarrow T_{\rightarrow}}{\Gamma \vdash \begin{array}{lccc} |_{j=1}^{r} & p_j & \mapsto & t_j \\ | & p & \mapsto & t \\ |_{k=r+1}^{m} & p_k & \mapsto & t_k \end{array} \mathrel{|\!:}_c S \rightarrow T_{\rightarrow}}
$$

$$
\textit{reorder} \qquad \frac{\Gamma \vdash \begin{array}{lccc} |_{i=1}^{k} & p_i & \mapsto & t_i \\ | & p & \mapsto & t \\ |_{j=k+1}^{m} & p_j & \mapsto & t_j \end{array} \mathrel{|\!:}_c S \rightarrow T_{\rightarrow} \quad k \neq k'}{\Gamma \vdash \begin{array}{lccc} |_{i=1}^{k'} & p_i & \mapsto & t_i \\ | & p & \mapsto & t \\ |_{j=k'+1}^{m} & p_j & \mapsto & t_j \end{array} \mathrel{|\!:}_c S \rightarrow T_{\rightarrow}}
$$

Figure 4.2: Complete pattern abstractions

$\rightarrow$; for example, $T_{\rightarrow}$. Sequential pattern abstractions can only be used to produce antecedents for the *pair* and the *record* rules; all other pattern abstractions must have the sequentializing removed.

We can use the *pair* rule to collapse the above example as follows:

$$z : B \vdash \left| \begin{array}{lcl} (y,x) & \mapsto & (y,x) \\ (y,x) & \mapsto & (x,y) \\ (y,x) & \mapsto & (x,x) \end{array} \right. \;\models_c A \times A \rightarrow A \times A \quad x,y,z \in \mathrm{var}$$

More examples of the use of the *pair* rule appear later.

The *unit* rule operates similarly to the *variable* rule; a unit pattern is associated with each complete term. The resulting pattern abstraction must be complete since pattern () will match any term of type 1.

Let $r :_c L(A)$ be a discriminant. A pattern abstraction that can match any possible r must, for one thing, have patterns for each constructor of type L. Furthermore, the group of patterns for each constructor must have nested patterns that can match any possible input of their type. This is the result the *constructor* rule generates. For example, suppose we have the following judgement:

$$x : A \times \mathit{list}(A) \vdash () \mapsto f\,x \models_c 1 \rightarrow T\,,\; y \mapsto f\,y \models_c A \times \mathit{list}(A) \rightarrow T \quad x,y \in \mathrm{var}$$

Using the *constructor* rule we can apply constructors of type *list* to get

$$x : A \times \mathit{list}(A) \vdash \left| \begin{array}{l} \mathit{nil}() \mapsto f\,x \models_c 1 \rightarrow T \\ \mathit{cons}\,y \mapsto f\,y \models_c \mathit{list}(A) \rightarrow T \end{array} \right. \quad x,y \in \mathrm{var}$$

For a more complicated example, suppose we have the judgement

$$x : A \vdash () \mapsto f\,x \models_c 1 \rightarrow T\,,\; \left| \begin{array}{lcl} (y,\mathit{nil}) & \mapsto & f\,y \\ (y,\_) & \mapsto & g\,y \end{array} \right. \models_c A \times \mathit{list}(A) \rightarrow T \quad x,y \in \mathrm{var}$$

The above judgement contains complete pattern abstractions for types $1 \rightarrow T$ and $A \times \mathit{list}(A) \rightarrow T$ so we can prefix the appropriate *list* constructors and combine the pattern abstractions to get

$$x : A \vdash \left| \begin{array}{lll} \mathit{nil}() & \mapsto & f\,x \\ \mathit{cons}(y, \mathit{nil}) & \mapsto & f\,y \\ \mathit{cons}(y, \_) & \mapsto & g\,y \end{array} \right. \;\models_c \mathit{list}(A) \to T \quad x, y \in \mathrm{var}$$

Each constructor of the datatype must have its own set of patterns, even if their domain types are the same. For instance, continuing with the above example, we add a unit pattern using the *unit* rule.

$$x : A \vdash \left| \begin{array}{lllll} () & \mapsto & \mathit{nil}() & \mapsto & f\,x \\ () & \mapsto & \mathit{cons}(y, \mathit{nil}) & \mapsto & f\,y \\ () & \mapsto & \mathit{cons}(y, \_) & \mapsto & g\,y \end{array} \right. \;\models_c 1 \to (\mathit{list}(A) \to T) \quad x, y \in \mathrm{var}$$

Given only the above judgement we could not use the *constructor* rule to add boolean constructors since there is only one pattern abstraction but two constructors, *false* and *true*. However, if we had the following judgement in hand:

$$x : A \vdash \begin{array}{l} \left| \begin{array}{lllll} () & \mapsto & \mathit{nil}() & \mapsto & f\,x \\ () & \mapsto & \mathit{cons}(y, \mathit{nil}) & \mapsto & f\,y \\ () & \mapsto & \mathit{cons}(y, \_) & \mapsto & g\,y \end{array} \right. \;\models_c 1 \to (\mathit{list}(A) \to T)\,, \\ \\ \left| \begin{array}{lllll} () & \mapsto & \mathit{cons}(z, \_) & \mapsto & f\,z \\ () & \mapsto & \mathit{nil}() & \mapsto & g\,z \end{array} \right. \;\models_c 1 \to (\mathit{list}(A) \to T) \end{array} \quad x, y, z \in \mathrm{var}$$

Then we can apply the boolean constructors to get

$$x : A \vdash \left| \begin{array}{lllll} \mathit{false}() & \mapsto & \mathit{nil}() & \mapsto & f\,x \\ \mathit{false}() & \mapsto & \mathit{cons}(y, \mathit{nil}) & \mapsto & f\,y \\ \mathit{false}() & \mapsto & \mathit{cons}(y, \_) & \mapsto & g\,y \\ \mathit{true}() & \mapsto & \mathit{cons}(z, \_) & \mapsto & f\,z \\ \mathit{true}() & \mapsto & \mathit{nil}() & \mapsto & g\,z \end{array} \right. \;\models_c \mathit{bool} \to (\mathit{list}(A) \to T) \quad x, y, z \in \mathrm{var}$$

Since each constructor must be applied to every pattern phrase in a complete pattern abstraction the resulting pattern abstraction must also be complete. Using the *pair* rule the above becomes:

$$x : A \vdash \left| \begin{array}{lcl} (\mathit{false}(), \mathit{nil}()) & \mapsto & f\,x \\ (\mathit{false}(), \mathit{cons}(y, \mathit{nil})) & \mapsto & f\,y \\ (\mathit{false}(), \mathit{cons}(y, \_)) & \mapsto & g\,y \\ (\mathit{true}(), \mathit{cons}(z, \_)) & \mapsto & f\,z \\ (\mathit{true}(), \mathit{nil}()) & \mapsto & g\,z \end{array} \right. \models_c \mathit{bool} \times \mathit{list}(A) \to T \quad x, y, z \in \mathrm{var}$$

Pattern abstractions formed using the *pair* rule are always complete since they arose from a pattern abstraction that was already complete.

As one would expect, the *record* rule operates similarly to the *pair* rule. The pattern abstraction in the antecedent of the *record* rule must have a pattern for each first order destructor in each phrase (in the same order). These are then simply collapsed into a record pattern. The declaration context must contain a variable for each higher order destructor. Recall the stack datatype from chapter 1. Assume we have the following judgement

$$\Gamma,\ f : \mathit{nat} \to \mathit{stack}(\mathit{nat}) \vdash \left| \begin{array}{lcl} (\_, \mathit{succ}\,\mathit{zero}) & \mapsto & 1 \\ (\_, \mathit{zero}) & \mapsto & 2 \\ (\_, \_) & \mapsto & 3 \end{array} \right. \models_c \mathit{stack}(\mathit{nat}) \to T$$

Then we can derive the following using the *record* rule.

$$\Gamma,\ f : \mathit{nat} \to \mathit{stack}(\mathit{nat}) \vdash \left| \begin{array}{lcl} (\mathit{push} : f, \mathit{pop} : (\_, \mathit{succ}\,\mathit{zero})) & \mapsto & 1 \\ (\mathit{push} : f, \mathit{pop} : (\_, \mathit{zero})) & \mapsto & 2 \\ (\mathit{push} : f, \mathit{pop} : (\_, \_)) & \mapsto & 3 \end{array} \right. \models_c \mathit{stack}(\mathit{nat}) \to T$$

An integer pattern abstraction is complete if every integer matches at least one pattern in the abstraction. Similarly for character patterns.

Once a complete pattern abstraction is constructed it is not possible to make it incomplete by adding more pattern phrases since all matches that were possible originally would still be possible. The *weaken* rule is useful for inserting pattern phrases with incomplete right hand sides into complete pattern abstractions.

A complete pattern abstraction cannot be made incomplete by changing the order of its constituent pattern phrases since any match that is possible with one ordering of pattern phrases is possible with any other ordering. The *reorder* rule allows the order of pattern phrases to be changed. It is used in conjunction with the *contructor replace* rule that is discussed in the next section.

4.2.1 Replacement rules

The replacement rules in figure 4.3 make it possible to replace patterns with variables. The replacement rules do not compromise the completeness of a pattern abstraction since the inserted variables will match any discriminant the replaced patterns would have matched.

The rhs of a pattern phrase operated on by the replacement rules may have sequential patterns. The variable used to replace the pattern cannot be present in the sequential patterns of the rhs since this would allow patterns with repeated variables to be generated. This requirement is given in the rules by $v \notin patt(t)$.

None of the variables in the pattern to be replaced can be present in the rhs of the pattern phrase since the pattern is removed by the replacement rules. The first antecedent, $\Gamma \vdash v \mapsto t \mid\!:_c T' \to T$, ensures this is the case. This requirement also confirms v as a legitimate pattern for t.

In addition, the pattern to be replaced must have been substituted, as a term, for the replacing variable in t. Since the replacing variable may have been used in another context in Π and Π' this enables the replacing variable to be placed in the rhs term and take on the value of the pattern. For exposition purposes the substitution notation is used to convey this requirement but, in practice, the substitution is actually an abstraction. An algorithm that links substitutions and abstractions is given in section 5.1.1.

unit, int or *char replace*

$$\frac{\begin{array}{l} \Gamma \vdash |v \mapsto t \mid:_c S \to T_\to \\ \Gamma \vdash \left| \begin{array}{c} \Pi \\ p \mapsto t \quad \mid:_c S \to T_\to \\ \Pi' \end{array}\right. \end{array} \qquad \begin{array}{l} p \text{ is a unit, int or char pattern} \\ v \in \mathrm{var}(S) \\ v \notin \mathit{patt}(t) \end{array}}{\Gamma \vdash \left| \begin{array}{c} \Pi \\ v \mapsto t \quad \mid:_c S \to T_\to \\ \Pi' \end{array}\right.}$$

constructor replace

$$\frac{\begin{array}{l} \Gamma \vdash |v \mapsto t \mid:_c L(A) \to T_\to \\ \Gamma \vdash \left| \begin{array}{c} \Pi \\ |_{i=1}^{n} c_i\, v_i \mapsto t[c_i\, v_i/v] \mid:_c L(A) \to T_\to \\ \Pi' \end{array}\right. \end{array} \qquad \begin{array}{l} v_i \in \mathrm{var}(E_i(A, L(A))) \\ v \in \mathrm{var}(L(A)) \\ v \notin \mathit{patt}(t) \end{array}}{\Gamma \vdash \left| \begin{array}{c} \Pi \\ v \mapsto t \quad \mid:_c L(A) \to T_\to \\ \Pi' \end{array}\right.}$$

pair replace

$$\frac{\begin{array}{l} \Gamma \vdash |v \mapsto t \mid:_c S \times S' \to T_\to \\ \Gamma \vdash \left| \begin{array}{c} \Pi \\ (v_0, v_1) \mapsto t[(v_0, v_1)/v] \quad \mid:_c S \times S' \to T_\to \\ \Pi' \end{array}\right. \end{array} \qquad \begin{array}{l} v_0 \in \mathrm{var}(S) \\ v_1 \in \mathrm{var}(S') \\ v \in \mathrm{var}(S \times S') \\ v \notin \mathit{patt}(t) \end{array}}{\Gamma \vdash \left| \begin{array}{c} \Pi \\ v \mapsto t \quad \mid:_c S \times S' \to T_\to \\ \Pi' \end{array}\right.}$$

record replace

$$\frac{\begin{array}{l} \Gamma \vdash |v \mapsto t \mid:_c R(A) \to T_\to \\ \Gamma \vdash \left| \begin{array}{c} \Pi \\ (d_i{:}v_i, d_j{:}v_j) \mapsto t[(d_i{:}v_i, d_j{:}v_j)/v] \mid:_c R(A) \to T_\to \\ \Pi' \end{array}\right. \end{array} \qquad \begin{array}{l} i = 1..r \quad j = r+1..n \\ v_i \in \mathrm{var}(F_i(A, R(A))) \\ v_j \in \mathrm{var}(E_j(A) \to F_j(A, (R(A)))) \\ v \in \mathrm{var}(R(A)) \quad v \notin \mathit{patt}(t) \end{array}}{\Gamma \vdash \left| \begin{array}{c} \Pi \\ v \mapsto t \quad \mid:_c R(A) \to T_\to \\ \Pi' \end{array}\right.}$$

Figure 4.3: Replacement rules

Finally, note that the *constructor replace* rules replaces n pattern phrases, one for each constructor of the datatype, with one pattern phrase. While it would be sufficient to have a rule that replaced only one constructor pattern, the rule given here more closely parallels what actually happens during the translation to core term logic.

The utility of these rules is best shown by example. We will derive the judgement below using the *constructor replace* rule. For the duration of this example, $f \in$ comb and $v, x, y \in$ var.

$$\Gamma\,,\; v : \mathit{list}(A) \vdash \left| \begin{array}{lcl} (\mathit{nil}, x) & \mapsto & v \\ (v', \mathit{true}) & \mapsto & f\ v \\ (\mathit{cons}\,y, \mathit{false}) & \mapsto & y \end{array} \right. \models_c \mathit{list}(A) \times \mathit{bool} \to \mathit{list}(A)$$

The *constructor replace* rule will be used to insert the variable pattern v'. Without this rule a constructor would necessarily be in its place. We will assume we have the following judgements:

$$\Gamma\,,\; v : \mathit{list}(A) \vdash \begin{array}{l} \left| \begin{array}{l} () \mapsto (x \quad \mapsto v) \\ x \ \mapsto (\mathit{true} \mapsto \{v' \mapsto f\ v\}\mathit{nil}\,x) \end{array} \right. \models_c 1 \to (\mathit{bool} \to \mathit{list}(A))\,, \\ \left| \begin{array}{l} y \mapsto (\mathit{true} \mapsto \{v' \mapsto f\ v\}\mathit{cons}\,y) \\ y \mapsto (\mathit{false} \mapsto y) \end{array} \right. \models_c A \times \mathit{list}(A) \to (\mathit{bool} \to \mathit{list}(A))\,, \\ \mid v' \mapsto (\mathit{true} \mapsto f\ v) \models_c \mathit{list}(A) \to (\mathit{bool} \to \mathit{list}(A)) \end{array}$$

Using the *constructor* rule we derive:

$$\Gamma\,,\; v : \mathit{list}(A) \vdash \left| \begin{array}{ll} \mathit{nil} & \mapsto (x \quad \mapsto v) \\ \mathit{nil}\,x & \mapsto (\mathit{true} \mapsto \{v' \mapsto f\ v\}\mathit{nil}\,x) \\ \mathit{cons}\,y & \mapsto (\mathit{true} \mapsto \{v' \mapsto f\ v\}\mathit{cons}\,y) \\ \mathit{cons}\,y & \mapsto (\mathit{false} \mapsto y) \end{array} \right. \models_c \mathit{list}(A) \to (\mathit{bool} \to \mathit{list}(A))$$

Now, using the *constructor replace* rule, we replace a constructor pattern with variable pattern, $v' \in \mathrm{var}(\mathit{list}(A))$.

$$\Gamma\,,\; v : \mathit{list}(A) \vdash \left|\begin{array}{lllll} nil() & \mapsto & x & \mapsto & v \\ v' & \mapsto & true() & \mapsto & f\,v \\ cons\,y & \mapsto & false() & \mapsto & y \end{array}\right. \quad \models_c \mathit{list}(A) \to (\mathit{bool} \to \mathit{list}(A))$$

Finally, the *pair* rule is applied:

$$\Gamma\,,\; v \vdash \begin{array}{lll} (nil(), x) & \mapsto & v \\ (v', true()) & \mapsto & f\,v \\ (cons\,y, false()) & \mapsto & y \end{array} \models_c \mathit{list}(A) \times \mathit{bool} \to \mathit{list}(A)$$

Note that the *reorder* rule can be used to group constructor pattern phrases prior to applying the *constructor replace* rule. This is especially useful when there are more than two constructors involved and, also, when the replacement variable pattern phrase needs to be placed last in the pattern abstraction.

Chapter 5

Translating the patterned term logic

The translation of patterned term logic programs to core term logic programs is patterned after the patterned term logic type theory given in the preceding two chapters. The type theory for core term logic is given in appendix A. The patterned term logic type theory constructed valid objects in the patterned term logic. The translation unwinds this construction step by step. If the translation can successfully reach basic judgements then the program will be known to be complete and, as a byproduct of this process, the core term logic structure of the program will be created.

Programs derived from patterned term logic type theory given in the last two chapters are correctly formed, well typed, complete, and contain no unbound variables. Input to the translation is guaranteed to be well formed as it will have been successfully parsed. Furthermore, it will be well typed and be without unbound variables having successfully passed the typechecker (Vesely, 1997). However, it may not be complete. The translation algorithm defined in this chapter ensures that the program is complete; incomplete programs are caught and reported at this stage.

Recall from section 3.1.5 that programs can either be function definitions

$$\text{def } f\{g_i\} = g \qquad \text{def } f = g$$

or executable terms. The translation of programs is given by

$$[\![\text{ def } f\{g_i\} = g]\!] \quad = \quad \text{def } f\{g_i\} = v \mapsto [\![\{g\}v]\!]_\tau$$

$$[\![\text{ def } f = g]\!] \quad = \quad \text{def } f = v \mapsto [\![\{g\}v]\!]_\tau$$

$$[\![t]\!] \quad = \quad [\![t]\!]_\tau$$

There are three major subcomponents of the program translation function. These are the term translation, $[\![\_]\!]_\tau$, given in section 5.1; the function translation, $[\![\_]\!]_\lambda$, given in section 5.3; and the pattern abstraction translation, $[\![\_]\!]_\alpha$, given in section 5.1. Normally, subscripts will not be shown on a translation function since the kind of translation can be determined by the type of its input.

We need to add a null term judgement to the core term logic type theory to facilitate the translation.

$$\frac{\Gamma \quad T \in types}{\Gamma \vdash NULL : T}$$

The null term is used as a placeholder during translation. Valid results of a program translation will contain no null terms. If null terms persist then it indicates incompleteness.

5.1 Pattern abstraction translation

The pattern abstraction translation algorithm, $[\![\_]\!]_\alpha$, takes as input a representation of a pattern abstraction and a list of core term logic terms, the discriminants, as input. The list of discriminants will be described using the SML (Paulson, 1991) notation for lists, that is, $r :: rs \equiv cons(r, rs)$ when r and rs are of the correct type. All terms in the representation of the pattern abstraction are core terms that may be incomplete and so may contain null terms. This representation may contain sequential patterns; the notation for these follows the conventions established in chapter 4; that is,

$$|_{i=1}^{m} p_i^1 \mapsto \ldots \mapsto p_i^{m_i} \mapsto t \not| S_1 \rightarrow (S_2 \rightarrow \ldots \rightarrow T).$$

The pattern abstraction translation unwinds a pattern abstraction with respect to the rules for generating complete pattern abstractions given in chapter 4. The unwinding proceeds until there are no patterns left to process. There are two situations where this may occur. In the first the input to the translation may contain no pattern phrases. In this case a null term is returned:

$$[\![|_{i=0}, \_]\!] \quad = \quad NULL$$

This type of call to the translation occurs during the processing of incomplete patterns and corresponds to the situation where there are no patterns that match a particular discriminant.

In the second case the input pattern abstraction may consist of a series of terms with no associated patterns. This occurs when the translation of the patterns is finished; all patterns have been translated and all that remain are terms. This is shown below.

$$[\![|_{i=1}^{m} t_i, \_]\!] = fatten(|_{i=1}^{m} t_i)$$

The function $fatten$ replaces null terms in t_i with the term t_{i+1}. This process begins at the bottom of the list where term t_m is left as is. If there is at least one complete term in the list, a complete term will be generated. The intuition for this is illustrated by the following example. Recall the case term example used in chapter 4.

$$\left\{\begin{array}{lll} s_1\, x & \mapsto & \left\{\begin{array}{lll} a_2 & \mapsto & h\, a_2 \end{array}\right\} f\, x \\ s_2\, x & \mapsto & \left\{\begin{array}{lll} a_1 & \mapsto & h\, a_1 \end{array}\right\} g\, x \\ \_ & \mapsto & \mathit{ff} \end{array}\right\} a \;.$$

After the inner case terms are translated into core term logic the translation of the outer pattern abstraction in the case function is:

$$\left[\!\!\left[\begin{array}{lll} s_1\, x & \mapsto & \left\{\begin{array}{lll} a_1 & \mapsto & NULL \\ a_2 & \mapsto & h\, a_2 \\ a_3 & \mapsto & NULL \end{array}\right\} f\, x \\ s_2\, x & \mapsto & \left\{\begin{array}{lll} a_1 & \mapsto & h\, a_1 \\ a_2 & \mapsto & NULL \\ a_3 & \mapsto & NULL \end{array}\right\} g\, x \\ \_ & \mapsto & \mathit{ff} \end{array} , [a] \right]\!\!\right] \;.$$

The first step in this translation is to build a case statement to discriminate between constructors s_1 and s_2. The translation proceeds by gathering together the pattern phrases that match on identical input. Eventually the translation call

$$\left[\!\!\left[\begin{array}{l} \left\{\begin{array}{lll} a_1 & \mapsto & NULL \\ a_2 & \mapsto & h\, a_2 \\ a_3 & \mapsto & NULL \end{array}\right\} f\, x \\ \mathit{ff} \end{array} , [\,] \right]\!\!\right]$$

will be generated since these two terms correspond to patterns that match input prefixed with the s_1 constructor. The function $fatten$ will use the ff term to fill in the null terms. This leaves

$$\left\{ \begin{array}{lcl} a_1 & \mapsto & f\!f \\ a_2 & \mapsto & h\,a_2 \\ a_3 & \mapsto & f\!f \end{array} \right\} f\,x$$

This is what we would expect. Any term that matches $s_1\,x$ will cause $f\,x$ to be executed. If the output of $f\,x$ is not a_2 then $f\!f$ must be returned, since the term that matched $s_1\,x$ will also match the don't care pattern. Similarly, for constructor s_2, the translation call

$$\left[\!\!\left[\begin{array}{|l} \left\{ \begin{array}{lcl} a_1 & \mapsto & h\,a_1 \\ a_2 & \mapsto & NULL \\ a_3 & \mapsto & NULL \end{array} \right\} g\,x \\ f\!f \end{array}, [\,] \right]\!\!\right]$$

generates

$$\left\{ \begin{array}{lcl} a_1 & \mapsto & h\,a_1 \\ a_2 & \mapsto & f\!f \\ a_3 & \mapsto & f\!f \end{array} \right\} g\,x$$

The result of the case term translation is

$$\left\{ \begin{array}{lcl} s_1\,x & \mapsto & \left\{ \begin{array}{lcl} a_1 & \mapsto & f\!f \\ a_2 & \mapsto & h\,a_2 \\ a_3 & \mapsto & f\!f \end{array} \right\} f\,x \\ s_2\,x & \mapsto & \left\{ \begin{array}{lcl} a_1 & \mapsto & h\,a_1 \\ a_2 & \mapsto & f\!f \\ a_3 & \mapsto & f\!f \end{array} \right\} g\,x \\ \_ & \mapsto f\!f & \end{array} \right\} a \quad .$$

which corresponds to the semantics of the original patterned term logic case term.

The rest of the section deals with the translation of pattern abstractions that do contain patterns.

5.1.1 Substitutions

Substituting a core term logic term, $r : U$, for a variable, $v \in \mathrm{var}(U)$ into $p_1 \mapsto t \mid\!\!: S \to T_{\to}$ is defined recursively as:

$$\begin{aligned} & (p_1 \mapsto t)[r/v] \mid\!\!: S_1 \to T_{\to} \\ = \quad & p_1 \mapsto t[r/v] \mid\!\!: S_1 \to T_{\to} \end{aligned}$$

$$\begin{aligned} & p \mapsto t[r/v] \mid\!\!: S \to T \\ = \quad & p \mapsto \{v \mapsto t\}r \mid\!\!: S \to T \end{aligned}$$

where the base case is a non-sequential term. Note the substitution is actually done using a core term logic abstraction. In practice, if the variable, v, has a don't care variable as an ancestor then the substitution is not performed since it is known that the v is not in the term t.

Function variable substitutions are complicated by the core term logic's lack of a facility for substituting function variables. Thus these substitutions must be done "manually" during the translation. If $g : R \to R'$ and $f \in \mathrm{var}(R \to R')$ then

$$\begin{aligned} & (p_1 \mapsto t)[g/f] \mid\!\!: S_1 \to T_{\to} \\ = \quad & p_1 \mapsto t[g/f] \mid\!\!: S_1 \to T_{\to} \end{aligned}$$

$$\begin{aligned} & (p \mapsto t)[g/f] \mid\!\!: S \to T \\ = \quad & p \mapsto t' \mid\!\!: S \to T \end{aligned}$$

where t' has had every instance of f replaced by g.

5.1.2 Variable patterns and unit patterns

If all the leading patterns of the pattern abstraction are variables then the head of the discriminant list is substituted for the variables. That is

$$[\![|_{i=1}^{m} v_i \mapsto t_i, r :: rs]\!] = [\![|_{i=1}^{m} t_i[r/v_i], rs]\!]$$

This corresponds to the *variable* rule of the completeness portion of the patterned term logic type theory (see figure 4.2).

If all the leading patterns are unit patterns then the patterns and the first discriminant on the list are discarded. That is

$$[\![|_{i=1}^{m} () \mapsto t_i, r :: rs]\!] = [\![|_{i=1}^{m} t_i, rs]\!]$$

This corresponds to the *unit* rule of the completeness portion of the patterned term logic type theory.

5.1.3 Constructor patterns

If the leading patterns of the pattern abstraction are of type $L(A)$, and at least one of the patterns is a constructor pattern, then a core term logic case term is constructed. A call to $[\![\_]\!]_\alpha$ is made with a modified pattern abstraction and discriminant list for each constructor of datatype L. Figure 5.1 shows how this is accomplished.

The function $[\![\_]\!]_{c_i}$ strips the constructor name from the lead pattern if the constructor name is c_i. If the constructor name is something other than c_i then the pattern phrase is dropped from the translation. This ensures that only terms that can be reached through constructor c_i are kept in the translation. If the lead pattern is a variable pattern then the discriminant r is substituted for the variable. A new variable pattern is introduced to take the substituted variable's place. In practice,

$$
\begin{array}{lll}
& [\![|_{k=1}^{m}\, p_k \mapsto t_k, r :: rs]\!] & p_k : L(A) \\
= & \left\{ {}_{i=1}^{n} c_i\, v_i \mapsto \left[\!\!\left[|_{k=1}^{m}\, [\![p_k \mapsto t_k, r]\!]_{c_i}, v_i :: rs \right]\!\!\right] \right\} r : T & v_i \in \mathrm{var}(E_i(A, L(A))) \\
& & v_i \text{ is new}
\end{array}
$$

$$
\begin{array}{ll}
& [\![c_i\, p \mapsto t, \_]\!]_{c_i} \\
= & p \mapsto t \mid: E_i(A, L(A)) \to T_{\to}
\end{array}
$$

$$
\begin{array}{lll}
& [\![v \mapsto t, r]\!]_{c_i} & v \in \mathrm{var}(L(A)) \\
& & v' \in \mathrm{var}(E_i(A, L(A))) \\
= & v' \mapsto t[v/r] \mid: E_i(A, L(A)) \to T_{\to} & v' \text{ is new}
\end{array}
$$

Figure 5.1: Pattern Abstraction Translation: Constructors

the new variable pattern is a don't care pattern. Since substitutions are not done with don't care patterns this prevents needless abstractions in the resulting core term logic term.

As an example of the translation of a pattern abstraction with constructors, consider once again the translation call

$$
\left[\!\!\left[\left|
\begin{array}{lll}
s_1\, x & \mapsto & \left\{ \begin{array}{lll} a_1 & \mapsto & NULL \\ a_2 & \mapsto & h\, a_2 \\ a_3 & \mapsto & NULL \end{array} \right\} f\, x \\
s_2\, x & \mapsto & \left\{ \begin{array}{lll} a_1 & \mapsto & h\, a_1 \\ a_2 & \mapsto & NULL \\ a_3 & \mapsto & NULL \end{array} \right\} g\, x \\
\_ & \mapsto & \mathit{ff}
\end{array} \right. , [a] \right]\!\!\right] .
$$

A case term is constructed:

$$\left\{\begin{array}{lll} s_1\, v_1 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} [\![s_1\, x \mapsto \{\ldots\}\, f\, x]\!]_{s_1} \\ [\![s_2\, x \mapsto \{\ldots\}\, g\, x]\!]_{s_1} \\ [\![\_ \mapsto \mathit{ff}]\!]_{s_1} \end{array}\right., [v_1]\right]\!\!\right] \\ s_2\, v_2 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} [\![s_1\, x \mapsto \{\ldots\}\, f\, x]\!]_{s_2} \\ [\![s_2\, x \mapsto \{\ldots\}\, g\, x]\!]_{s_2} \\ [\![\_ \mapsto \mathit{ff}]\!]_{s_2} \end{array}\right., [v_2]\right]\!\!\right] \end{array}\right\} a$$

The $\ldots$ represent the pattern abstractions of the inner case terms. The $[\![\_]\!]_{s_1}$ function calls are evaluated next:

$$[\![s_1\, x \mapsto \{\ldots\}\, f\, x]\!]_{s_1} = x \mapsto \{\ldots\}\, f\, x$$

Here the pattern $s_1\, x$ had the s_1 constructor so the pattern phrase was retained. The variable x is the nested pattern. In the next call,

$$[\![s_2\, x \mapsto \{\ldots\}\, g\, x]\!]_{s_1} = x \mapsto \{\ldots\}\, g\, x$$

the constructors do not match so this pattern phrase is dropped. The final call,

$$[\![\_ \mapsto \mathit{ff}]\!]_{s_1} = \_ \mapsto \mathit{ff},$$

has a lead variable pattern. However, it is a don't care variable and no substitution is done. The "new" variable pattern that is introduced is also a don't care pattern.

After the initial subsidiary function calls are evaluated the case term has the following form:

$$\left\{\begin{array}{lll} s_1\,v_1 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} x \mapsto \{\ldots\}\,f\,x \\ \_ \mapsto \mathit{ff} \end{array}\right., [v_1]\right]\!\!\right] \\ s_2\,v_2 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} x \mapsto \{\ldots\}\,g\,x \\ \_ \mapsto \mathit{ff} \end{array}\right., [v_2]\right]\!\!\right] \end{array}\right\}\,a$$

Both of the recursive calls to the pattern abstraction translation consist entirely of variable patterns. Thus we get

$$\left\{\begin{array}{lll} s_1\,v_1 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} \{x \mapsto \{\ldots\}\,f\,x\}v_1 \\ \mathit{ff} \end{array}\right., [\,]\right]\!\!\right] \\ s_2\,v_2 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} \{x \mapsto \{\ldots\}\,g\,x\}v_2 \\ \mathit{ff} \end{array}\right., [\,]\right]\!\!\right] \end{array}\right\}\,a$$

At this point all that remains is to return the final terms. This gives:

$$\left\{\begin{array}{lll} s_1\,v_1 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} \left\{x \mapsto \left\{\begin{array}{lll} a_1 & \mapsto & \mathit{ff} \\ a_2 & \mapsto & h\,a_2 \\ a_3 & \mapsto & \mathit{ff} \end{array}\right\}\,f\,x\right\}\,v_1 \\ \mathit{ff} \end{array}\right., [\,]\right]\!\!\right] \\ s_2\,v_2 & \mapsto & \left[\!\!\left[\left|\begin{array}{l} \left\{x \mapsto \left\{\begin{array}{lll} a_1 & \mapsto & h\,a_1 \\ a_2 & \mapsto & \mathit{ff} \\ a_3 & \mapsto & \mathit{ff} \end{array}\right\}\,g\,x\right\}\,v_2 \\ \mathit{ff} \end{array}\right., [\,]\right]\!\!\right] \end{array}\right\}\,a$$

Note the unnecessary variable substitution of v_1 for x and v_2 for x. Section 5.3 explains how this can be detected and avoided.

To complete the description of the above example we show the translation

$$[\![a_2() \mapsto h\, a_2, [f\, x]]\!]$$

This results in the case term

$$\left\{\begin{array}{lcl} a_1\, v_1 & \mapsto & \left[\!\!\left[[\![a_2() \mapsto h\, a_2]\!]_{a_1}, [v_1] \right]\!\!\right] \\ a_2\, v_2 & \mapsto & \left[\!\!\left[[\![a_2() \mapsto h\, a_2]\!]_{a_2}, [v_2] \right]\!\!\right] \\ a_3\, v_3 & \mapsto & \left[\!\!\left[[\![a_3() \mapsto h\, a_2]\!]_{a_3}, [v_3] \right]\!\!\right] \end{array}\right\} f\, x$$

The constructor functions are evaluated next. This gives:

$$\left\{\begin{array}{lcl} a_1\, v_1 & \mapsto & [\![\ \ , [v_1]]\!] \\ a_2\, v_2 & \mapsto & [\![() \mapsto h\, a_2, [v_2]]\!] \\ a_3\, v_3 & \mapsto & [\![\ \ , [v_3]]\!] \end{array}\right\} f\, x$$

The recursive functions for a_1 and a_3 resolve to null terms. The other function gives the result:

$$[\![() \mapsto h\, a_2, [v_2]]\!] = [\![h\, a_2, [\,]]\!] = h\, a_2$$

Thus the final case term looks like this:

$$\left\{\begin{array}{lcl} a_1\, v_1 & \mapsto & NULL \\ a_2\, v_2 & \mapsto & h\, a_2 \\ a_3\, v_3 & \mapsto & NULL \end{array}\right\} f\, x$$

The constructor part of the translation has corresponding rules in the completeness portion of the type theory. If all the patterns are constructor patterns then the corresponding rule is the *constructor* rule (see figure 4.2). The *constructor* rule brings pattern abstractions together; the translation reverses the process. If there are vari-

$$
\begin{array}{cl}
 & [\![|_{k=1}^{m}\ p_k \mapsto t_k, r :: rs]\!] \\
= & \left[\!\!\left[|_{k=1}^{m}\ [\![p_k \mapsto t_k, r]\!]_{\times}, p_0(r) :: p_1(r) :: rs \right]\!\!\right]
\end{array}
\quad p_k : S \times S'
$$

$$
\begin{array}{cl}
 & [\![(p, p') \mapsto t, \_]\!]_{\times} \\
= & p \mapsto p' \mapsto t \mid: S \to (S' \to T_{\to})
\end{array}
$$

$$
\begin{array}{cl}
 & [\![v \mapsto t, r]\!]_{\times} \\
= & v_0 \mapsto v_1 \mapsto t[r/v] \mid: S \to (S' \to T_{\to})
\end{array}
\quad
\begin{array}{l}
v_0 \in \mathrm{var}(S) \\
v_1 \in \mathrm{var}(S') \\
v_0, v_1 \text{ are new}
\end{array}
$$

Figure 5.2: Pattern Abstraction Translation: Pairs

able patterns present, along with the constructor patterns, then the *inductive replace* rule is also required (see figure 4.3). It consolidates a group of particular pattern phrases, one for each constructor, into one pattern phrase with a variable pattern. Again, this is the reverse process of the translation.

5.1.4 Pair patterns

If some of the leading patterns of a pattern abstraction are pair patterns then the pairs are sequentialized and the translation called with the altered pattern abstraction. Figure 5.2 shows how this is done.

The function $[\![\_]\!]_{\times}$ sequentializes pair patterns and substitutes discriminants for variables. In the case of a variable pattern it introduces two new variables to the pattern phrase as placeholders. As with the constructor translation, in practice the introduced variables are don't care variables. The lead discriminant on the list of discriminants is replaced with two destructed versions; one for p_0 and one for p_1.

$$
\begin{array}{lll}
& [\![|_{k=1}^{m}\ p_k \mapsto t_k, r :: rs]\!] & i = 1..r \\
= & \left[\!\!\left[|_{k=1}^{m}\ [\![p_k \mapsto t_k, r]\!]_{R(A)}, d_i(r) :: rs \right]\!\!\right] & j = r+1..n \\
& & p_k : R(A) \text{ or } p_k : 1
\end{array}
$$

$$
\begin{array}{lll}
& & i = 1..r \\
& [\![(d_i : p_i, d_j : v_j) \mapsto t, r]\!]_{R(A)} & j = r+1..n \\
= & p_i \mapsto t[\{x_j \mapsto d_j(x_j, r)\}/v_j] \mathrel{\ddagger} F_i(A, R(A)) \to T_{\to} & x_j \in \mathrm{var}(E_j(A)) \\
& & x_j \text{ is new}
\end{array}
$$

$$
\begin{array}{lll}
& [\![v \mapsto t, r]\!]_{R(A)} & i = 1..r \\
& & v' \in \mathrm{var}(R(A)) \\
= & v' \mapsto t[r/v] \mathrel{\ddagger} F_i(A, R(A)) \to T_{\to} & v' \text{ is new}
\end{array}
$$

Figure 5.3: Pattern Abstraction Translation: Records

An example translation of a pair pattern abstraction appears in the next subsection and is similar to the translation for record patterns.

The pair translation corresponds to the *pair* and *pair replace* rules of the completeness portion of the type theory.

5.1.5 Record patterns

If the leading patterns of the pattern abstraction have type $R(A)$ and at least one is a record pattern then the process illustrated in figure 5.3 is used. The patterns of the first order destructors are sequentialized and appropriate substitutions are made for the function variable patterns of the higher order destructors. The discriminant r is replaced on the discriminant list with a series of terms corresponding to the destruction of r by the first order destructors of R.

When passed a record pattern phrase the function $[\![\_]\!]_{R(A)}$ sequentializes the patterns of the first order destructors. Function variables of higher order destructors are substituted immediately. The abstraction $\{x_j \mapsto d_j(x_j, r)\}$ is used to replace the higher order destructor variables since r may not be in record form (and will thus need to be destructed to obtain the required function).

When passed a variable pattern phrase the function $[\![\_]\!]_{R(A)}$ removes the variable pattern and replaces it with a sequence of new variables; one for each first order destructor of the type R. The discriminant is substituted for the original variable in the term of the phrase.

Recall the example presented in section 4.2 of the *stack* record pattern abstraction. Its translation for some discriminant r is:

$$\left[\!\!\left[\left| \begin{array}{lll} (push : f, pop : (\_, 1)) & \mapsto & f\,1 \\ (push : f, pop : (\_, 0)) & \mapsto & f\,2 \\ y & \mapsto & y \end{array} \right. , [r] \right]\!\!\right]$$

The translation sequentializes the *pop* first order destructor patterns and substitutes for the *push* higher order destructor patterns.

$$\left[\!\!\left[\left| \begin{array}{lll} (\_, 1) & \mapsto & f\,1[\{x_1 \mapsto push(x_1, r)\}/f] \\ (\_, 0) & \mapsto & f\,2[\{x_2 \mapsto push(x_2, r)\}/f] \\ v_1 & \mapsto & y[r/y] \end{array} \right. , [pop(r)] \right]\!\!\right]$$

The abstraction function substituted for the function variable serves to bind the input to the higher order destructor to a known variable. This facilitates the substitution process (see section 5.1.1). When the substitutions are done we have:

$$\left[\!\!\left[\begin{array}{|lll} (\_,1) & \mapsto & \{x_1 \mapsto push(x_1,r)\}1 \\ (\_,0) & \mapsto & \{x_2 \mapsto push(x_2,r)\}2 \\ v_1 & \mapsto & \{y => y\}r \end{array}, [pop(r)]\right]\!\!\right]$$

We will take this one step further to illustrate how pair pattern abstractions are translated:

$$\left[\!\!\left[\begin{array}{|lllll} \_ & \mapsto & 1 & \mapsto & \{x_1 \mapsto push(x_1,r)\}1 \\ \_ & \mapsto & 0 & \mapsto & \{x_2 \mapsto push(x_2,r)\}2 \\ v_2 & \mapsto & v_3 & \mapsto & \{y => y\}r[pop\, r/v_1] \end{array}, [p_0(pop\, r), p_1(pop\, r)]\right]\!\!\right]$$

The corresponding rules from the completeness portion of the type theory are the *record* and the *coinductive replace* rules.

5.1.6 Integer and character patterns

Since integers and characters are not true datatypes there are no case combinators available for them. Instead the $\_<\_$ relation that is available for these datatypes is used to partition the pattern phrases. The $\_<\_$ relation takes the discriminant, r, and some partitioning value, q, as input. The $\_<\_$ relation is used exclusively for partitioning because this builds a structure that lends itself to optimization using the theories presented in (Cockett & Zhu, 1989). However these optimizations have yet to be implemented.

We will only discuss the process for translating integer pattern abstractions here since translating character pattern abstractions is very similar. The rules for translating integer pattern abstractions are given in figure 5.4. Rules are given for character pattern abstractions in figure 5.5.

$$\llbracket |_{k=1}^{m}\, p_k \mapsto t_k, r :: rs \rrbracket \qquad p_k : int$$

$$= \left\{ \begin{array}{lll} true & \mapsto & \left\llbracket |_{k=1}^{m} \llbracket p_k \mapsto t_k, q \rrbracket_{<}, [r] \right\rrbracket \\ false & \mapsto & \left\llbracket |_{k=1}^{m} \llbracket p_k \mapsto t_k, q \rrbracket_{\geq}, [r] \right\rrbracket \end{array} \right\} r < q : T \qquad \begin{array}{l} \text{for some } p_k = i..j, \\ q = i \text{ if } i \in \mathbb{Z} \\ q = j+1 \text{ if } j \in \mathbb{Z} \end{array}$$

$$\llbracket v \mapsto t, \_ \rrbracket_{<} = \llbracket v \mapsto t, \_ \rrbracket_{\geq} = v \mapsto t \mid\!: int \to T_{\to} \quad v \in \text{var}$$

$q \leq i$	$\llbracket i..j \mapsto t, q \rrbracket_{<}$	pattern phrase is dropped
$q > i$ and $q \leq j+1$	$\llbracket -\infty..j \mapsto t, q \rrbracket_{<} = v \mapsto t \mid\!: int \to T_{\to}$	$v \in \text{var}(int)$ v is new
$q > i$ and $q \leq j+1$	$\llbracket i..j \mapsto t, q \rrbracket_{<} = i..\infty \mapsto t \mid\!: int \to T_{\to}$	
$q > j+1$	$\llbracket i..j \mapsto t, q \rrbracket_{<} = i..j \mapsto t \mid\!: int \to T_{\to}$	

$q > j$	$\llbracket i..j \mapsto t, q \rrbracket_{\geq}$	pattern phrase is dropped
$q \geq i$ and $q \leq j$	$\llbracket i..\infty \mapsto t, q \rrbracket_{\geq} = v \mapsto t \mid\!: int \to T_{\to}$	$v \in \text{var}(int)$ v is new
$q \geq i$ and $q \leq j$	$\llbracket i..j \mapsto t, q \rrbracket_{\geq} = -\infty..j \mapsto t \mid\!: int \to T_{\to}$	
$q < i$	$\llbracket i..j \mapsto t, q \rrbracket_{\geq} = i..j \mapsto t \mid\!: int \to T_{\to}$	

Figure 5.4: Pattern Abstraction Translation: Integers

$$
\begin{array}{ll}
\llbracket |_{k=1}^{m} \; p_k \mapsto t_k, r :: rs \rrbracket & p_k : \mathit{char} \\
= \left\{ \begin{array}{l} \mathit{true} \mapsto \left\llbracket |_{k=1}^{m} \; \llbracket p_k \mapsto t_k, q \rrbracket_{<}, [r] \right\rrbracket \\ \mathit{false} \mapsto \left\llbracket |_{k=1}^{m} \; \llbracket p_k \mapsto t_k, q \rrbracket_{\geq}, [r] \right\rrbracket \end{array} \right\} r < q : T & \begin{array}{l} \text{for some } p_k = c..c', \\ q = c \text{ if } c \neq \texttt{\x0} \\ q = c' + 1 \text{ if } c' \neq \texttt{\x8f} \end{array}
\end{array}
$$

$$
\llbracket v \mapsto t, \_ \rrbracket_{<} = \llbracket v \mapsto t, \_ \rrbracket_{\geq} = v \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} \quad v \in \mathrm{var}
$$

$$
\begin{array}{lll}
q \leq c & \llbracket c..c' \mapsto t, q \rrbracket_{<} \text{ pattern is dropped} & \\
q > c \text{ and } q \leq c' + 1 & \llbracket \texttt{\x0}..c' \mapsto t, q \rrbracket_{<} = v \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} & \begin{array}{l} v \in \mathrm{var}(\mathit{char}) \\ v \text{ is new} \end{array} \\
q > c \text{ and } q \leq c' + 1 & \llbracket c..c' \mapsto t, q \rrbracket_{<} = c..\texttt{\x8f} \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} & \\
q > c' + 1 & \llbracket c..c' \mapsto t, q \rrbracket_{<} = c..c' \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} &
\end{array}
$$

$$
\begin{array}{lll}
q > c' & \llbracket c..c' \mapsto t, q \rrbracket_{\geq} \text{ pattern is dropped} & \\
q \geq c \text{ and } q \leq j & \llbracket c..\texttt{\x8f} \mapsto t, q \rrbracket_{\geq} = v \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} & \begin{array}{l} v \in \mathrm{var}(\mathit{char}) \\ v \text{ is new} \end{array} \\
q \geq c \text{ and } q \leq j & \llbracket c..c' \mapsto t, q \rrbracket_{\geq} = \texttt{\x0}..c' \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} & \\
q < c & \llbracket c..c' \mapsto t, q \rrbracket_{\geq} = c..c' \mapsto t \mathrel{|\!:} \mathit{char} \to T_{\to} &
\end{array}
$$

Figure 5.5: Pattern Abstraction Translation: Characters

A value q is chosen from the integer patterns with the restriction that q cannot be $-\infty$ or ∞. With characters, q cannot be `\x0` or `\x8f` and so there is a special case where a suitable q cannot be found. In this case all character range patterns are converted to new variable patterns.

The value q is used to split the pattern phrases into two groups; one group matches any value less than q (the true case), the other group matches any value greater than or equal to q (the false case). Pattern phrases are dropped or their patterns are modified based on their relation to q.

Consider the translation

$$\left[\!\!\left[\left| \begin{array}{lll} -\infty..5 & \mapsto & t_1 \\ 7..10 & \mapsto & t_2 \\ 2..\infty & \mapsto & t_3 \end{array} \right., [r] \right]\!\!\right]$$

We choose $q = 5+1$ from the first pattern phrase. Constructing the case term we get

$$\left\{ \begin{array}{lll} \mathit{true} & \mapsto & \left[\!\!\left[\left| \begin{array}{l} [\![-\infty..5 \mapsto t_1, 6]\!]_< \\ [\![7..10 \mapsto t_2, 6]\!]_< \\ [\![2..\infty \mapsto t_3, 6]\!]_< \end{array} \right., [r] \right]\!\!\right] \\ \mathit{false} & \mapsto & \left[\!\!\left[\left| \begin{array}{l} [\![-\infty..5 \mapsto t_5, 6]\!]_\geq \\ [\![7..10 \mapsto t_2, 6]\!]_\geq \\ [\![2..\infty \mapsto t_3, 6]\!]_\geq \end{array} \right., [r] \right]\!\!\right] \end{array} \right\} r < 6$$

For the *true* case above, $r \leq 5$. Thus r will always match the first pattern phrase and never be able to match the second pattern phrase. The third pattern phrase will match only if $r \geq 2$. Since the r always matches the first pattern it can be replaced with a variable.

For the *false* case above, $r \geq 6$. Here r will never match the first pattern phrase.

The second pattern phrase remains unchanged but the integer pattern of the third pattern phrase can be changed to a variable pattern since it will always match r. After this processing the case term becomes:

$$\left\{\begin{array}{lll} \mathit{true} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_1 & \mapsto & t_1 \\ 2..\infty & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \\ \mathit{false} & \mapsto & \left[\!\!\left[\begin{array}{|lll} 7..10 & \mapsto & t_2 \\ v_2 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \end{array}\right\} r < 6$$

Doing the *true* case translation first we set $q = 2$ and get:

$$\left\{\begin{array}{lll} \mathit{true} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_1 & \mapsto & t_1 \end{array}, [r]\right]\!\!\right] \\ \mathit{false} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_1 & \mapsto & t_1 \\ v_3 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \end{array}\right\} r < 2$$

Now, doing the *false* case from the first case term we set $q = 7$ and get:

$$\left\{\begin{array}{lll} \mathit{true} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_2 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \\ \mathit{false} & \mapsto & \left[\!\!\left[\begin{array}{|lll} -\infty..10 & \mapsto & t_2 \\ v_2 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \end{array}\right\} r < 7$$

The final step is the *false* case above. $q = 11$ and we get:

$$\left\{\begin{array}{lll} \mathit{true} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_2 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \\ \mathit{false} & \mapsto & \left[\!\!\left[\begin{array}{|lll} v_4 & \mapsto & t_2 \\ v_2 & \mapsto & t_3 \end{array}, [r]\right]\!\!\right] \end{array}\right\} r < 11$$

Putting the results of all the subtranslations together we get the following core term logic case term:

$$\left\{ \begin{array}{lll} \mathit{true} & \mapsto & \left\{ \begin{array}{lll} \mathit{true} & \mapsto & \left\{ v_1 \;\mapsto\; t_1 \right\} r \\ \mathit{false} & \mapsto & \left\{ v_1 \;\mapsto\; t_1 \right\} r \end{array} \right\} r < 2 \\ \mathit{false} & \mapsto & \left\{ \begin{array}{lll} \mathit{true} & \mapsto & \left\{ v_2 \;\mapsto\; t_3 \right\} r \\ \mathit{false} & \mapsto & \left\{ \begin{array}{lll} \mathit{true} & \mapsto & \left\{ v_2 \;\mapsto\; t_3 \right\} r \\ \mathit{false} & \mapsto & \left\{ v_4 \;\mapsto\; t_2 \right\} r \end{array} \right\} r < 11 \end{array} \right\} r < 7 \end{array} \right\} r < 6$$

For $r \leq 5$, the case term produces t_1; $7 \leq r \leq 10$ produces t_2; and for $r = 6$ or $r \geq 11$ we get t_3. This is the correct meaning of the patterned term logic pattern abstraction. Note the inner case term with discriminant $r < 2$ is not required since both the *true* and *false* results are identical. This indicates that optimization would be beneficial. Also note that the translation of integer patterns must eventually finish since at least one of the integer values of the leading patterns is removed with each iteration.

5.2 Translating terms

The translation function for terms, $[\![\_]\!]_\tau$ is given in figure 5.6. Given the patterned term logic type theory definition of terms, figure 3.2, the translation, for the most part, is a straightforward recursion on the components of the terms.

The exceptions are pattern abstractions associated with higher-order destructors in record terms. Here, a case term is created with a new variable, v, as input. It is more convenient to use the case function translation to check pattern abstractions for completeness than to perform this check every place a pattern abstraction is encountered. In addition, the variable v is needed because the pattern abstraction translation requires both a pattern abstraction and its discriminant as input.

variable	$[\![v]\!] = v \quad v \in \mathrm{var}$	
unit	$[\![()]\!] = ()$	
pair	$[\![(t_1, t_2)]\!] = ([\![t_1]\!], [\![t_2]\!])$	
record	$[\![(d_i : t_i, d_j : g_j)]\!] = (d_i : [\![t_i]\!], d_j : v \mapsto [\![\{g_j\}v]\!])$	$i = 1..r$ $j = r+1..n$ $v \in varSet(E_j(A))$ v is new
application	$[\![f\,t]\!] = [\![f]\!]_\lambda\,[\![t]\!]$	

Figure 5.6: Term Translation

Examples of term translations are found in the next section.

5.3 Translating Functions

The translation for functions, $[\![\_]\!]_\lambda$, is given in figure 5.7. Like the term translation, the function translation follows the patterned term logic type theory definition of functions given in figure 3.4.

The technique used in the term translation for dealing with pattern abstractions is also employed here in the translations for folds, maps, unfolds, and defined functions.

The pattern abstraction of the case function may be required to be complete. If so, the term returned by pattern abstraction translation is tested to see if it contains null terms by the function *isComplete*. If none are found *isComplete* returns the term, otherwise it generates an error message and the translation halts. The pattern abstraction function has a dash on it to indicate that the terms of the pattern phrases *function variable* $$[\![f : S \to T]\!] = f \quad f \in \mathrm{var}$$

cstr, dstr or def function $$[\![s : S \to T]\!] = s \quad s \in \mathrm{cstr} \cup \mathrm{dstr} \cup \mathrm{comb}([\,]\,,\, S \to T)$$

def function $$[\![f\{|_{i=1}^{m}\, g_i \,|\!\!:_c S_i \to T_i, \}]\!] = f\{|_{i=1}^{m}\, v_i \mapsto [\![\{g_i\}v_i,]\!]\} \qquad \begin{array}{l} v_i \in \mathrm{var}(S_i) \\ v_i \text{ is new} \end{array}$$

case $$[\![\{g\}]\!] = isComplete(\{v \mapsto [\![g, v]\!]'_{\alpha}\}) \quad g \,|\!\!:_c S \to T \quad v \in \mathrm{var}(S) \quad v \text{ is new}$$

case (incomplete) $$[\![\{g\}]\!] = \{v \mapsto [\![g, v]\!]'_{\alpha}\} \quad g \,|\!\!: S \to T \quad v \in \mathrm{var}(S) \quad v \text{ is new}$$

fold $$[\![\{\!|\,|_{i=1}^{n} c_i : g_i \,|\!\}]\!] = \{\!|\,|_{i=1}^{n} c_i : v_i \mapsto [\![\{g_i\}v_i]\!]\,|\!\} \quad v_i \in \mathrm{var}(E_i(A, L(A))) \quad v_i \text{ is new}$$

map $$\left[\!\!\left[T \left\{ \begin{array}{c} g_i^{+}, \\ g_j^{-}, \\ g_k^{+} \;\&\; g_k^{-}, \end{array} \right\} \right]\!\!\right] = T \left\{ \begin{array}{l} v_i \mapsto \left[\!\!\left[\{g_i^{+}\}v_i\right]\!\!\right], \\ v_j \mapsto \left[\!\!\left[\{g_j^{-}\}v_j\right]\!\!\right], \\ v_k \mapsto \left[\!\!\left[\{g_k^{+}\}v_k\right]\!\!\right] \& \; v_k \mapsto \left[\!\!\left[\{g_k^{-}\}v_k\right]\!\!\right] \end{array} \right\}$$

$$\text{where} \quad \begin{array}{lll} i = 1..p & v_i \in \mathrm{var}(A_i) & v_i \text{ is new} \\ j = p+1..q & v_j \in \mathrm{var}(A_j) & v_j \text{ is new} \\ k = q+1..r & v_k \in \mathrm{var}(A_k) & v_k \text{ is new} \end{array}$$

unfold

$$\left[\!\!\left[\left(\Big|_{k=1}^{m} \; p_k \mapsto \begin{array}{l} |_{i=1}^{r} \; d_i : t_i^k \\ |_{j=r+1}^{n} \; d_j : g_j^k \end{array} \right) : S \to R(A) \right]\!\!\right]$$

$$= \left(v \mapsto \begin{array}{l} |_{i=1}^{r} \; d_i : |_{k=1}^{m} \; \left[\!\!\left[\{p_k \mapsto t_i^k\}v\right]\!\!\right] \\ |_{j=r+1}^{n} \; d_j : v_j \mapsto |_{k=1}^{m} \; \left[\!\!\left[\{p_k \mapsto \{g_j^k\}v_j\}v\right]\!\!\right] \end{array} \right)$$

$$\text{where} \quad \begin{array}{l} |_{k=1}^{m} \; p_k \mapsto t_i^k \;\; |\!\!:_c S \to F_i(A, R(A)) \\ |_{k=1}^{m} \; \{p_k \mapsto \{g_j^k\}v_j\}v \;\; |\!\!:_c S \to F_j(A, R(A)) \\ v \in \mathrm{var}(S) \quad v \text{ is new} \\ v_j \in \mathrm{var}(E_j(A)) \quad v_j \text{ is new} \end{array}$$

Figure 5.7: Function Translation

in the pattern abstraction must be translated to core term logic before calling $[\![\_]\!]_\alpha$ proper.

A call to the defined function, $f \in \mathrm{comb}([A \to A], S \to T)$, is translated below:

$$\begin{aligned}
& [\![f\{x \mapsto x\}]\!] \\
= \quad & f\{v \mapsto [\![\{x \mapsto x\}v]\!]\} \\
= \quad & f\{v \mapsto [\![\{x \mapsto x\}]\!]\,[\![v]\!]\} \\
= \quad & f\{v \mapsto isComplete(\{v' \mapsto [\![\{x \mapsto x\}, v']\!]\})v\} \\
= \quad & f\{v \mapsto isComplete(\{v' \mapsto \{x \mapsto x\}v'\})v\} \\
= \quad & f\{v \mapsto \{v' \mapsto \{x \mapsto x\}v'\}v\}
\end{aligned}$$

The above example illustrates a small problem with the translation of case terms as described. Abstractions may be unnecessarily introduced into the translated term. This can be remedied by checking the term returned by the translation whenever a variable is introduced. If the term is a case term and its discriminant is the same variable that was introduced then the new variables can be discarded and the abstraction of the case term used instead. In step 5 of the above translation this would have resulted in

$$f\{v \mapsto isComplete(\{x \mapsto x\})v\}$$

leading to the final result of

$$f\{x \mapsto x\}.$$

The benefit of introducing a variable when translating a case term is that the case term discriminant is abstracted to a variable. This means the discriminant, which may be arbitrarily complex, is only evaluated once by the abstract machine. If the initial patterns were pairs or records then destructors would be applied to the discriminant during the pattern abstraction translation and it would necessarily be evaluated more than once.

The translation of the unfold function requires some explanation. In the core term logic unfold function only one iteration of destructors is used and state information is contained in one variable (v in the figure). Thus, the translation must move the outer patterns of the patterned term logic unfold function in front of the destructors and use a case term to pattern match on the value of the state. For first order destructors this is all that is needed. For higher order destructors, the usual trick of introducing a variable and converting the abstraction to a case term is performed first. Note that since the new case term is ensconced in the case term that introduces the state, the new case term does not need to be complete (although the outer case term must be).

We illustrate the translation process of an unfold using the stack implemenation given in chapter 3.

$$\left[\!\!\left[\left(\begin{array}{lcl} nil & \mapsto & \left|\begin{array}{lcl} push & : & a \mapsto [a] \\ pop & : & ([\,], x) \end{array}\right. \\ cons(a,l) & \mapsto & \left|\begin{array}{lcl} push & : & a' \mapsto cons(a', cons(a,l)) \\ pop & : & (l,a) \end{array}\right. \end{array}\right)\right]\!\!\right]$$

$$= \left(v \mapsto \begin{array}{lcl} pop & : & \left[\!\!\left[\left\{\begin{array}{lcl} nil & \mapsto & ([\,], x) \\ cons(a,l) & \mapsto & (l,a) \end{array}\right\} v\right]\!\!\right] \\ push & : & v' \mapsto \left[\!\!\left[\left\{\begin{array}{lcl} nil & \mapsto & \{a \mapsto [a]\} v' \\ cons(a,l) & \mapsto & \{a' \mapsto cons(a', cons(a,l))\} v' \end{array}\right\} v\right]\!\!\right] \end{array}\right)$$

The variable v has been introduced as a state variable and each destructor has a translation of a case term generated by gathering the original outer patterns together in a case function and applying it to the new state variable. After the subtranslations are complete the end result is:

$$\left(v \mapsto \begin{array}{l} pop \ : \left\{\begin{array}{lcl} nil & \mapsto & ([\,], x) \\ cons\, x & \mapsto & \{l \mapsto \{a \mapsto (l,a)\} p_0\, x\} p_1\, x \end{array}\right\} v \\ push : v' \mapsto \left\{\begin{array}{ll} nil & \mapsto \{a \mapsto [a]\} v' \\ cons(a,l) & \mapsto \{l \mapsto \{a \mapsto \{a' \mapsto cons(a', cons(a,l))\} v'\} p_0\, x\} p_1\, x \end{array}\right\} v \end{array}\right)$$

Chapter 6

Future Work

This thesis has given a type theory for the patterned term logic and has shown how to translate its programs into core term logic. While there was no attempt to prove the translation correct, it was modeled closely on the term logic and therefore it is easy to see that the translation will detect incomplete programs and that it will terminate. The translation also ensures that complex discriminants will not be evaluated more than once and, with some common sense, it avoids introducing unnecessary abstractions.

However, the translation does not always generate optimal code. The average number of decisions required to reach a righthand side may not be optimal. One example of this was seen in section 5.1.6, an integer pattern abstraction translation, and another example was given in section 1.3.8 where translations were discussed. The papers (Simpson & Cockett, 1992) and (Cockett & Zhu, 1989) describe methods that can be used to optimize the number of decisions in a pattern abstraction translation.

Another optimization that could be made is to reduce the size of the generated core term logic program. Often identical subtrees are produced while translating pattern abstractions. While these subtrees are shared in the core term logic representation, later translations copy the subtrees. This increase in the size can be minimized by representing the identical subtrees as function calls and storing the code for the underlying function elsewhere. It would not be difficult to identify candidate subtrees; they are formed during constructor pattern abstraction translations when variable pattern phrases are put into each constructor's subtranslation, when $NULL$ terms are filled in by the function $fatten$ and during translation of integer and character patterns.

A third optimization could reduce the number of decisions required to reach a rhs. An abstraction optimized for the number of decisions will quite likely have paths to some right hand sides that are longer than others, that is, it may not be balanced. This may result in often used paths through the tree that are longer than necessary. This optimization would require the recompilation of the code after runtime experience and would only be appropriate if input to the abstraction did not have widely fluctuating values.

There are some improvements that could be made to the user interface. Repeated variables in **charity** patterns are not allowed (see section 1.3) only because, in general, it is not possible to determine if two terms are equal in **charity**. This is because some terms may be infinite. It would be possible to identity these potentially infinite terms by their types and repeated variables of these types could be disallowed. Types which allowed repeated variables would need built in equality functions; something that is currently not available.

Another useful feature would allow patterns to share a term, in other words, the pattern phrase would have more than one pattern associated with a term. It is not always possible to use a variable for this purpose. For example, a datatype with four constructors may have a pattern abstraction with one term for two of four possible constructor patterns and another term for the other two. Allowing variables in the patterns would complicate the implementation but it should be possible.

Currently only integers and characters have constructors that are ordered; this ordering makes it possible to use range patterns with them. In some situations it would be useful if constructors of user defined datatypes were ordered. Order could be implicitly given by the order of constructors in the datatype definition (although this could be the source of errors if the order was inadvertently changed) or it could be given explicitly through a syntactic extension to datatype definitions. A builtin function for each datatype that could determine the relative order of two constructors would also be required.

String patterns could be extended to use regular expressions in the manner of the perl programming language. This extension would be extremely useful for processing data from ascii files after **charity** is given input/output capabilities.

Finally, it is possible to simplify the patterned term logic and the core term logic. All builtin **charity** functions can be represented as folds or unfolds (Cockett & Fukushima, 1992). It is possible for the parser to translate builtin functions to unfolds and folds as it is building the data representation of a program. Thus the patterned term logic translation would not have to deal with cases, maps and records.

Appendix A

Core Term Logic Type Theory

The core term logic lacks the sophisticated patterns found in the patterned term logic. Nested constructor, function variable, pair, record and range patterns are not used. Completeness is enforced in the case function and abstraction function. The case function must have every constructor represented. The constructor patterns are simple, therefore, the nested value of any well-typed input will match. Abstraction patterns are either variables, in which case they match anything, or units, in which case there is only one element to match. Constructor patterns are defined along with the case function.

The following three figures define the core term logic.

variable	$$\frac{\Gamma \vdash \varphi \quad v \in \mathrm{var}(T) \quad v \text{ is not in } \Gamma}{\Gamma, v : T \vdash \varphi}$$
unit	$$\frac{\Gamma \vdash \varphi}{\Gamma\,,\,() : 1 \vdash \varphi}$$
abstraction	$$\frac{\Gamma\,,\,p : S \vdash t : T}{\Gamma \vdash p \mapsto t \mathrel{\vert\!:} S \to T}$$

Figure A.1: Core term logic: patterns

variable	$$\frac{\Gamma\,,\,v : T \vdash \varphi \quad v \in \mathrm{var}}{\Gamma\,,\,v : T \vdash v : T}$$
unit	$$\frac{}{\vdash () : 1}$$
pair	$$\frac{\Gamma \vdash t : T\,,\,t' : T'}{\Gamma \vdash (t, t') : T \times T'}$$
record	$$\frac{\Gamma \vdash \begin{array}{ll} t_i : F_i(A, R(A))\,, & i = 1..r \\ g_j \mathrel{\vert\!:} E_j(A) \to F_j(A, R(A)) & j = r+1..n \end{array}}{\Gamma \vdash (d_i : t_i, d_j : g_j) : R(A)}$$
application	$$\frac{\Gamma \vdash t : S\,,\,f : S \to T}{\Gamma \vdash f\,t : T}$$

Figure A.2: Core term logic: terms

structor or *defined function*

$$\frac{\Gamma \vdash \varphi \quad s : S \to T \in \mathrm{cstr} \cup \mathrm{dstr} \cup \mathrm{comb}([\,]\,,\, S \to T)}{\Gamma \vdash s : S \to T}$$

defined function

$$\frac{\Gamma \vdash |_{i=1}^{m}\, g_i \mathrel{|\!:} S_i \to T_i \quad f \in \mathrm{comb}([S_i \to T_i]\,,\, S \to T)}{\Gamma \vdash f\{_{i=1}^{m} g_i\} : S \to T}$$

abstraction function

$$\frac{\Gamma \vdash g \mathrel{|\!:} S \to T}{\Gamma \vdash \{g\} : S \to T}$$

case

$$\frac{|_{i=1}^{r}\ \Gamma\,,\, p_i : E_i(A, L(A)) \vdash t_i : T}{\Gamma \vdash \{_{i=1}^{n} c_i\, p_i \mapsto t_i\} : L(A) \to T}$$

fold

$$\frac{\Gamma \vdash |_{i=1}^{n}\, g_i \mathrel{|\!:} E_i(A,T) \to T \quad c_i \in \mathrm{cstr}(L)}{\Gamma \vdash \{|\,|_{i=1}^{n} c_i : g_i\,|\} : L(A) \to T}$$

map

$$\frac{\Gamma \vdash \begin{array}{ll} |_{i=1}^{p}\ g_i^{+} \mathrel{|\!:} A_i \to B_i, & i = 1..p \\ |_{j=p+1}^{q}\ g_j^{-} \mathrel{|\!:} B_j \to A_j, & j = p+1..q \\ |_{k=q+1}^{r}\ g_k^{+} \mathrel{|\!:} A_k \to B_k, & k = q+1..r \\ |_{k=q+1}^{r}\ g_k^{-} \mathrel{|\!:} B_k \to A_k & T \in \mathrm{types} \end{array}}{\Gamma \vdash T \left\{ \begin{array}{c} g_i^{+}, \\ g_j^{-}, \\ g_k^{+} \,\&\, g_k^{-} \end{array} \right\} : T(A) \to T(B)}$$

unfold

$$\frac{\Gamma\,,\, v : S \vdash \begin{array}{l} |_{i=1}^{r}\ t_i : F_i(A, R(A))\,, \\ |_{j=r+1}^{n}\ g_j \mathrel{|\!:} E_j(A) \to F_j(A, R(A)) \end{array} \quad v \in \mathrm{var}}{\Gamma \vdash \left(\left| v \mapsto \begin{array}{l} |_{i=1}^{r}\ d_i : t_i \\ |_{j=r+1}^{n}\ d_j : g_j \end{array} \right|\right) : S \to R(A)}$$

Figure A.3: Core term logic: functions

Appendix B

Glossary Of Functions And Datatypes

B.1 Charity builtin functions

Case function

Delimiters: $\{ \quad \}$

Introduced: section 1.1.2

Map function

Delimiters: $TYPE\ \{ \quad \}$

Introduced: section 1.1.2

Fold function

Delimiters: $\{| \quad |\}$

Introduced: section 1.1.2

Record function

Delimiters: $(\quad)$

Introduced: section 1.1.2

Unfold function

Delimiters: $(\!| \quad |\!)$

Introduced: section 1.1.2

B.2 Datatypes

***bool* datatype**

Definition: data $bool \to C = \begin{array}{lcl} \mathit{false} & : & 1 \to C \\ \mathit{true} & : & 1 \to C \end{array}$

Introduced: section 1.1.2

***list* datatype**

Definition: data $list(A) \to C = \begin{array}{lcl} \mathit{nil} & : & 1 \to C \\ \mathit{cons} & : & A \times C \to C \end{array}$

Introduced: section 1.1.2

***nat* datatype**

Definition: data $nat \to C = \begin{array}{lcl} \mathit{zero} & : & 1 \to C \\ \mathit{succ} & : & C \to C \end{array}$

Introduced: section 1.1.2

_SF_ datatype

Definition: data $SF(A) \rightarrow C = \begin{array}{lcl} \mathit{ff} & : & 1 \rightarrow C \\ ss & : & A \rightarrow C \end{array}$

Introduced: section 1.1.2

tree datatype

Definition: data $tree(A) \rightarrow C = \begin{array}{lcl} leaf & : & 1 \rightarrow C \\ node & : & A \times list(C) \rightarrow C \end{array}$

Introduced: section 2.3

inflist datatype

Definition: data $C \rightarrow infList(A) = \begin{array}{lcl} head & : & C \rightarrow A \\ tail & : & C \rightarrow A \times C \end{array}$

Introduced: section 1.1.2

Lprod datatype

Definition: data $C \rightarrow Lprod(A, B) = \begin{array}{lcl} fst & : & C \rightarrow A \\ snd & : & C \rightarrow B \end{array}$

Introduced: section 1.1.2

qstack datatype

Definition: data $S \rightarrow qstack(A, Q) = \begin{array}{lcl} qpop & : & S \rightarrow S \times (A \times Q) \\ qpush & : & S \rightarrow A \times S \end{array}$

Introduced: section 3.1.4

stack datatype

Definition: data $S \rightarrow stack(A) = \begin{array}{lcl} pop & : & S \rightarrow S \times A \\ push & : & S \rightarrow A \Rightarrow S \end{array}$

Introduced: section 1.1.2

Bibliography

Augustsson, L. (1985). Compiling Pattern Matching. In Jouannaud, J.-P. (Ed.), *LNCS*, Vol. 201, pp. 368–381. Functional Programming Languages and Computer Architecture, Springer-Verlag.

Cockett, J. R. B., & Zhu, Y. (1989). A new incremental learning technique for decision trees with thresholds. In *Proceedings of Applications of Artificial Intelligence VI*. SPIE 1989 Technical Symposium Southeast on Optics, Electro–Optics, and Sensors. Orlando, FL.

Cockett, R. (1992). Notes: Coinductive Data Types And Examples.. University of Calgary CPSC 501 class notes.

Cockett, R. (1993a). Charitable Thoughts.. University of Calgary CPSC 501 class notes in draft form.

Cockett, R. (1993b). CPSC 501: Advanced Programming Techniques.. University of Calgary CPSC 501 class notes.

Cockett, R., & Fukushima, T. (1992). about Charity. Tech. rep. #92/480/18, University of Calgary. Research Report.

Cockett, R., & Spencer, D. (1992a). Strong Categorical Datatypes I. In Seely, R. A. G. (Ed.), *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS.

Cockett, R., & Spencer, D. (1992b). Strong Categorical Datatypes II: A term logic for categorical programming. (to appear).

Fukushima, T., & Tuckey, C. (1996). *Charity User Manual*. Draft available from: `http://www.cpsc.ucalgary.ca/projects/charity/home.html`.

Holyer, I. (1991). *Functional Programming With Miranda*. Pitman Publishing, 128 Long Acre, London WC2E 9AN.

Laville, A. (1987). Lazy Pattern Matching In The ML Language. In Nori, K. V. (Ed.), *LNCS*, Vol. 287, pp. 400–419. Foundations Of Software Technology And Theoretical Computer Science, Springer-Verlag.

Paulson, L. C. (1991). *ML for the Working Programmer*. Cambridge University Press.

Peyton Jones, S. L., & Wadler, P. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International. From the Prentice-Hall International Series In Computer Science.

Schroeder, M. (1997). Higher-Order Charity. Master's thesis, University of Calgary.

Sekar, R. C., Ramesh, R., & Ramakrishnan, I. V. (1992). Adaptive Pattern Matching. Tech. rep., SUNY Stony Brook.

Simpson, T., & Cockett, R. (1992). Sequentializing Programs Defined By Pattern Matching. This paper is available by ftp. The address is: `ftp.cpsc.ucalgary.ca:/pub/projects/charity/PAPERS/PatMat.ps.Z`.

Vesely, P. M. (1997). Typechecking the Charity Term Logic. Notes on implementing type checking in charity (unpublished).

Yee, D. B. (1995). Implementing the Charity Abstract Machine. Master's thesis, University of Calgary.